OBJECT ORIENTED PROGRAMMING USING JAVA (CST 205)

Prepared by ***Prof.Renetha J.B. LMCST***

MODULE 1

Introduction

Module 1 - Topics

- Approaches to Software Design
- Object Modeling Using Unified Modeling Language (UML)
- Introduction to Java

Module 1 - Topics

- **Approaches to Software Design**
- Object Modeling Using Unified Modeling Language (UML)
- Introduction to Java

Approaches to Software Design

√ Functional Oriented Design

√ Object Oriented Design

√ Case Study of Automated Fire Alarm System.

Software

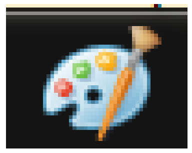
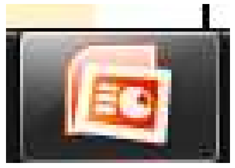
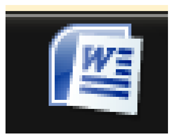
- Software
  - Software is a collection of instructions that *enable the user to interact with a computer*, its hardware or perform tasks.

- Without software, most computers would be useless.
  - For example, without Internet browser software, we could not surf the Internet.
  - Without an operating system, the browser could not run on your computer.

Software - types

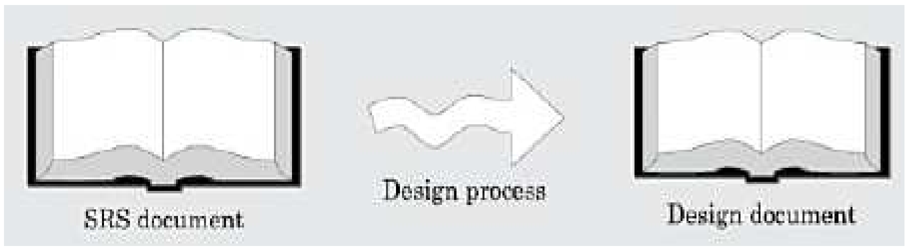
- There are two types of software
  1. System Software – E.g- Operating System, Compilers, Interpreters etc

  1. Application Software - Microsoft Word, MS paint, Microsoft PowerPoint etc.

Software Design

- During the software design phase,
  - the design document is produced, based on the customer requirements (documented in the SRS - Software Requirements Specification document)



- The **design process** essentially *transforms the SRS document into a design document.*

Software Design

- The design document produced at the end of the design phase should be **implementable using a programming language** in the subsequent (coding) phase.

- Software design is the first step in SDLC (Software Design Life Cycle)

APPROACHES TO SOFTWARE DESIGN

There are two fundamentally different approaches to software design —

- Function-oriented design,
- Object-oriented design.

Functional Oriented Design (FOD)

- In function-oriented design, the system is comprised of many smaller sub-systems known as **functions**.

- These functions are capable of performing significant task in the system.

Functional Oriented Design (FOD)- Features

The salient features of the function-oriented design approach:

- Top-down decomposition
- Centralised system state

FOD-Top-down decomposition

- A system at starting level(high level) is viewed as a black box that provides certain services (also known as **high-level functions**) to the users of the system.
- This function may be refined into the **sub-functions**.
- Each of these sub-functions may be split into <u>**more detailed subfunctions**</u> and so on.

FOD-Top-down decomposition

- Function oriented design inherits some properties of structured design where divide and conquer methodology is used(**top-down approach**).

FOD (Top down design - example)

- Consider a Library management System
- Some of the functions in this system are:
  - *create-new-library member*
  - *issue book*
  - *return book*

FOD (Top down design - example)

- Consider one function **create-new-library-member** to creates the record for a new member,
- This function may be refined into the following subfunctions:
  - *assign-membership-number*
  - *create-member-record*
  - *print-bill*

Each of these sub-functions may be split into more detailed subfunctions and so on.

FOD-Centralised system state

- The system state is centralised and shared among different functions.
- The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event.
- Such data in procedural programs usually have **global scope** and are **shared by many modules(functions)**.

FOD-Centralised system state(Eg.)

- For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system.

FOD-Centralised system state(Eg)

- For example, in the library management system, several functions ***share data*** such as *member-records* for reference and updation:
  - create-new-member
  - delete-member
  - update-member-record

Function-oriented design approaches

Some well-established function-oriented design approaches are :

- Structured design by Constantine and Yourdon
- Jackson's structured design by Jackson
- Warnier-Orr methodology
- Step-wise refinement by Wirth
- Hatley and Pirbhai's Methodology

Object oriented Design

- In the object oriented design approach, the system is viewed as **collection of objects** (i.e. entities).
- The state is **decentralized** among the objects and each object manages its own state information.- **no globally shared data**

Object oriented Design(contd)

- Each object is associated with a set of functions that are called its **methods**.
- Each object contains its own data and is responsible for managing it.
- The data internal to an object cannot be accessed directly by other objects
  - can be accessed only through invocation of the methods of the object.

Object oriented Design(contd)

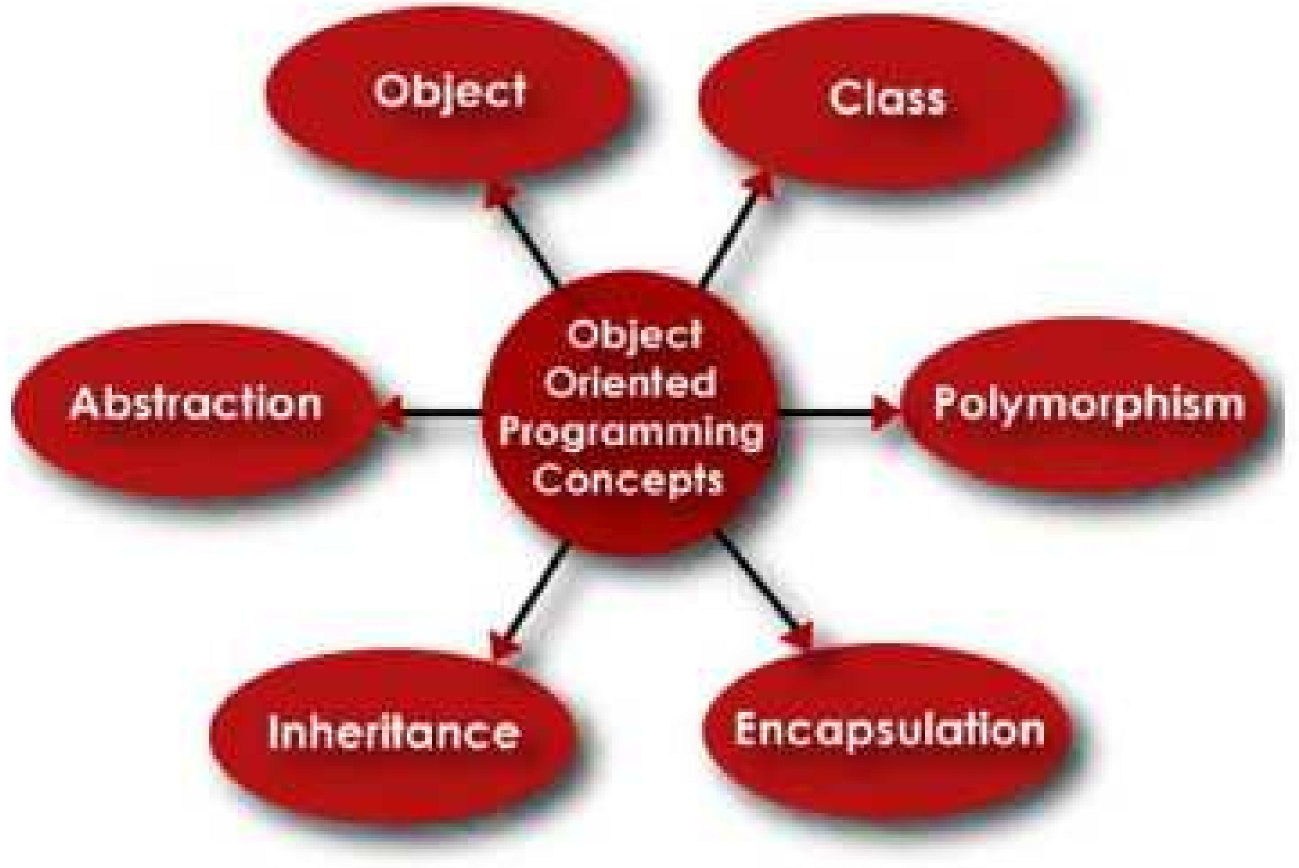
- The object-oriented design paradigm makes extensive use of the principles of **abstraction** and **decomposition.**
  - Objects decompose a system into functionally independent modules.
  - Objects can also be considered as instances of abstract data types (ADTs).

Object oriented Design(contd.)

Data abstraction(data hiding):

- The principle of data abstraction implies that how data is exactly stored is abstracted away.
- This means that any entity outside the object *will not have knowledge* about
  - how data is exactly stored, organised, and manipulated inside the object.
- The entities external to the object can <u>access the data internal to an object only by calling certain well-defined methods</u> supported by the object.

Basic Object Oriented concepts



Function Oriented Design	Object Oriented design
• The basic abstraction is the **services(functions)** that are available to the users of the system such as issue book ,display book details etc.	• The basic abstraction is **real-world entities(objects)** such as member, book, book-register etc.
• state information is available in a **centralised shared data store**.	• state information exists in the form of **data distributed** among several objects of the system
• Function oriented techniques group functions together (they constitute a higher level function)	• Object oriented techniques group functions together on the basis of the data they operate on.

Case Study of

Automated Fire Alarm System

Automated fire-alarm system—customer requirements

The owner of a large multi-storied building wants to have a computerised fire alarm system designed, developed, and installed in his building. <u>Smoke detectors and fire alarms</u> would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors.

Automated fire-alarm system—customer requirements

The owner of a large multi-storied building wants to have a computerised fire alarm system designed, developed, and installed in his building. Smoke detectors and fire alarms would be placed in each room of the building.

The fire alarm system would monitor the **status of these smoke detectors.**

Automated fire-alarm system(contd.)

- Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system **should determine the location** at which the fire has been sensed and then **sound the alarms only in the neighbouring locations.**
- The fire alarm system should also **flash an alarm message** on the computer console.

Automated fire-alarm system (contd.)

- Fire fighting personnel would man the console round the clock.
- After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the fire fighting personnel.

Fire Alarm system-Function oriented design

/* **Global data** (system state) accessible by various functions */

BOOL **detector_status [MAX_ROOMS];**

int **detector_locs [MAX_ROOMS];**

BOOL **alarm_status [MAX_ROOMS]**; /* *alarm activated when status is set */*

int **alarm_locs [MAX_ROOMS];** /* *room number where alarm is located*

int **neighbor alarm[MAX_ROOMS][10];** /* *each detector has at most 10 neighboring locations */*

Fire Alarm system-Function oriented design

The **functions** which operate on the system state are:

interrogate_detectors()

get_detector_location()

determine_neighbor()

ring_alarm()

reset_alarm()

report_fire_location()

Fire Alarm system-Object oriented design

class detector

attributes: status, location, neighbors

operations:

create, sense_status, get_location, find_neighbors

class alarm

attributes: location, status

operations:

create, ring_alarm, get_location, reset_alarm

class sprinkler

attributes: location, status

operations: create, activate-sprinkler, get_location, reset-sprinkler

Analysis

Function oriented approach

- The system state (data) is **centralised** and several functions access and modify this central data.
- Data is **global** and can be easily accessed.
- The basic unit of designing a function oriented program is functions and modules.
- Functions appear as verbs.
- TOP DOWN

Object oriented approach

- The state information (data) is **distributed** among various objects.
- Data is **private** in different objects and cannot be accessed by the other objects.
- The basic unit of designing an object oriented program is objects.
- Objects appear as nouns.
- BOTTOM UP

Reference Text Book

- Rajib Mall, Fundamentals of Software Engineering, 4th edition, PHI, 2014.

THANK YOU

OBJECT ORIENTED PROGRAMMING USING JAVA (CST 205)

Prepared by ***Prof.Renetha J.B. LMCST***

MODULE 1

Introduction

Topic: **Object Modeling Using Unified Modeling Language (UML)- Basic object oriented concepts**

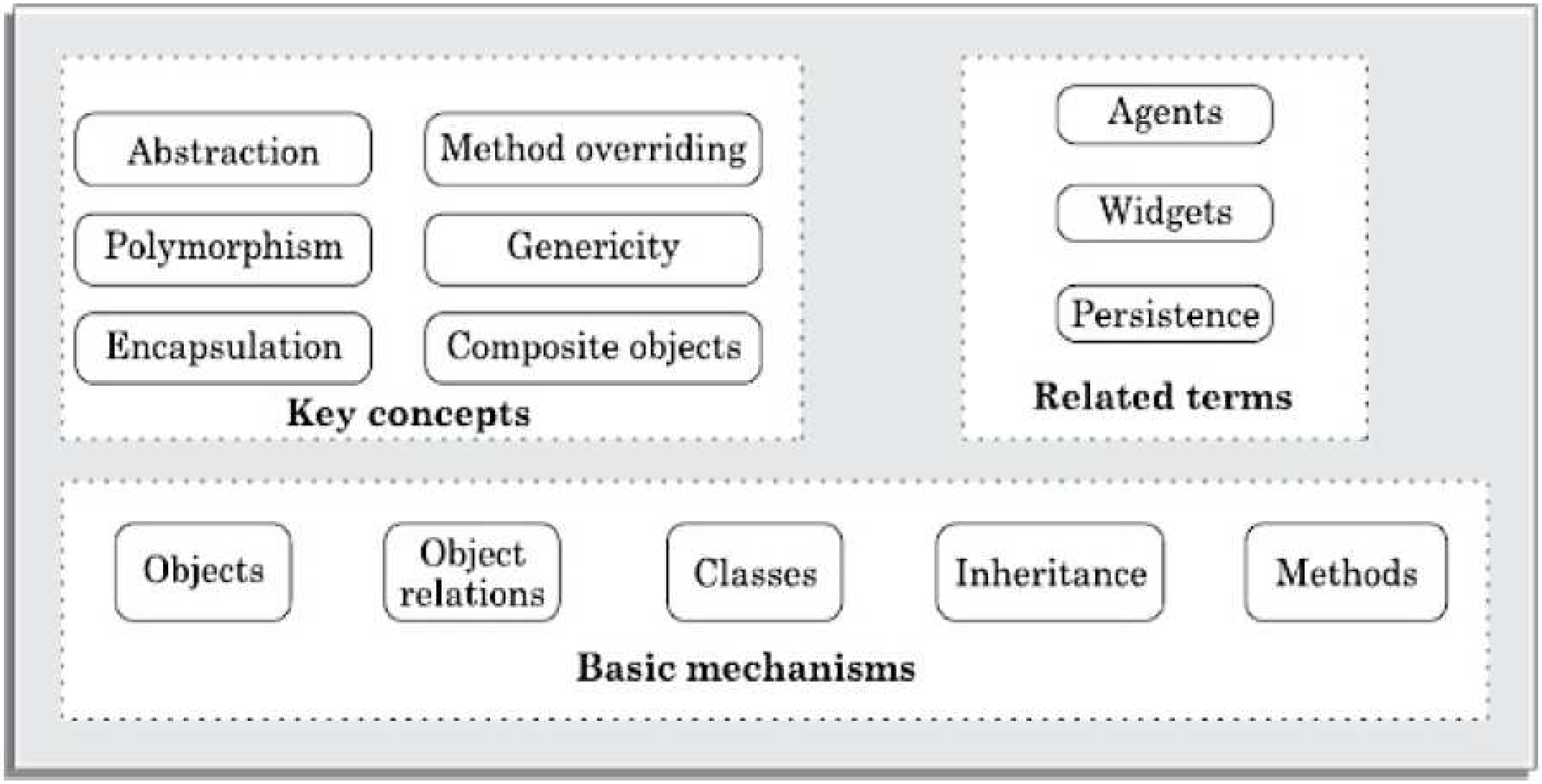
Model

- A model is constructed by **focusing only on a few aspects of the problem** and ignoring the rest.
- The model of a ***problem*** is called an ***analysis model.***
- The model of the ***solution*** (code) is called the ***design model***.
  - The design model is usually obtained by carrying out iterative refinements to the analysis model using a design methodology.

Modelling language:

- Modelling language: A modelling language consists of a **set of notations** using which design and analysis models are documented.
- A model can be documented using a modelling language such as Unified Modelling Language (UML).

BASIC OBJECT-ORIENTED CONCEPTS



OBJECT-ORIENTATION CONCEPTS

- Object
- Class
- Abstraction
- Encapsulation
- Class relationships
  - Inheritance
  - Association and link
  - Aggregation and composition
  - Dependency
- Polymorphism

Object

- Object in an object-oriented program usually represents
  - a *tangible* **real-world entity** *(can be touched)*
    - E.g.student, library member, a book, an issue register, etc.
  - or *conceptual* real-world entity
    - E,g. Loan, Job etc

Objects are real-world entities that has their own properties and behavior.

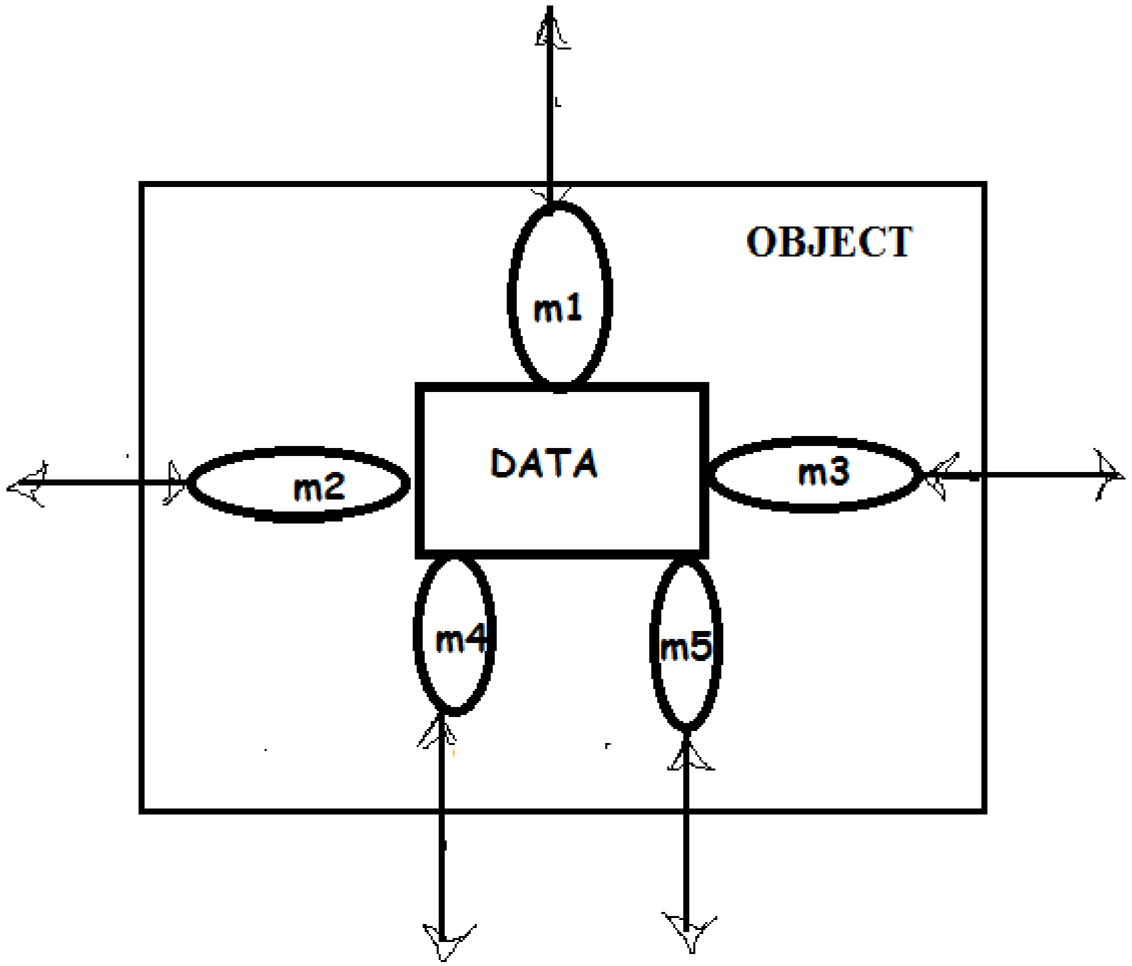
Object- Characteristics

- Each object essentially consists of
  - some **data** that is ***private*** to the object and
  - a set of **functions** (termed as operations or **methods**) that operate on those data.
- Each object hides its internal data from other objects.
- An object can access the private data of another object by invoking the methods supported by that object.

Object-Example

- Consider **Library Automation System**.
  - Objects can be library member, book, staff etc.
- Library Member object
  - private data
    - name of the member
    - membership number
    - address
  - methods
    - issue-book()
    - find-books-outstanding()
    - return-book()

Object



Here m1 , m2 etc are methods associated with object

Class

- Class is a **group of similar objects.**
- A class is a blueprint or prototype from which objects are created.
- A class is a generalized description of an object.
- An object is an instance of a class.

Class(contd.)

- All the objects in a class possess similar **attributes** (properties) and **methods** (behaviour or operation).

E.g Set of all students(objects) form **Student class**

Each student object possesses

- attributes(data)- Roll number, name etc.
- behaviour(methods)- study(), read(), write() etc.

Example

- E.g. Set of all library members(objects) would constitute the class LibraryMember
- Each LibraryMember object has
  – member name membershi number member address, etc.------> attributes(data)
  – issue-book(), return-book(), etc.

    ----> behaviour(methods)

ADT

- An **Abstract Data Type** is a **type** with associated operations, but its *representation(inner details) is hidden*.
- ADT is based on three concepts
  - **Abstract data-** data is hidden from outside. It can be only accessed through its methods.
  - **Data structure-c**onstructed from a collection of primitive data items.
  - **Data type-** In programming language theory, a data type identifies a group of variables having a particular behavious, and can be used to instantiate a variable. E.g.int c;

Class-ADT

Class is an ADT

- it has abstract data
- it has structure
- we can instantiate a class into objects.

Class and Object

- **Class** is just a **logical** definition.

- Consider Student class.
  - Let Sam is a student.
  - Sam is instance/object of the class Student

    Object has a **physical existence**

Methods

- *Function inside the class is called* ***method****.*
- The operations (such as create(), issue(), return(), etc.) supported by an object are implemented in the form of methods inside the class.

Difference between operation and method

- An **operation** is a specific responsibility of a class.
- The responsibility is ***implemented*** using a **method**..

Method overloading

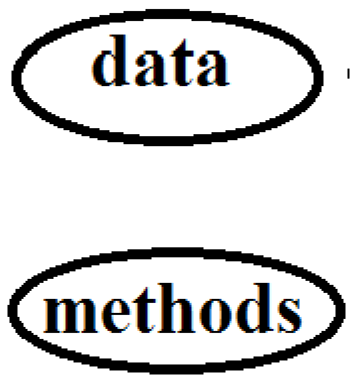
- In some cases the responsibility of a class can be implemented through **multiple methods** with the *same method name*. This is called **method overloading.**

Method overloading-Example

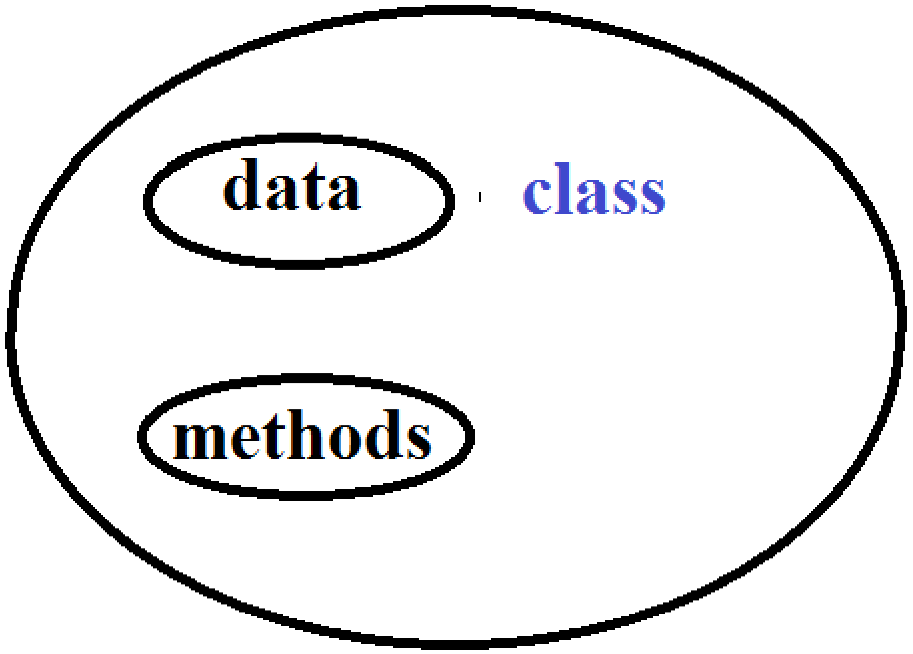
- E.g. Consider class named Circle
  - Assume that this class has three definitions for the create operation(method)—
    - int create()
      - *draws a circle with radius given inside create() function*
    - int create(int radius)
      - *draws a circle with radius passed to create() function*
    - int create(float x, float y, int radius);
      - *draws a circle with radius at specific position in xy coordinate passed to create() function*

Encapsulation

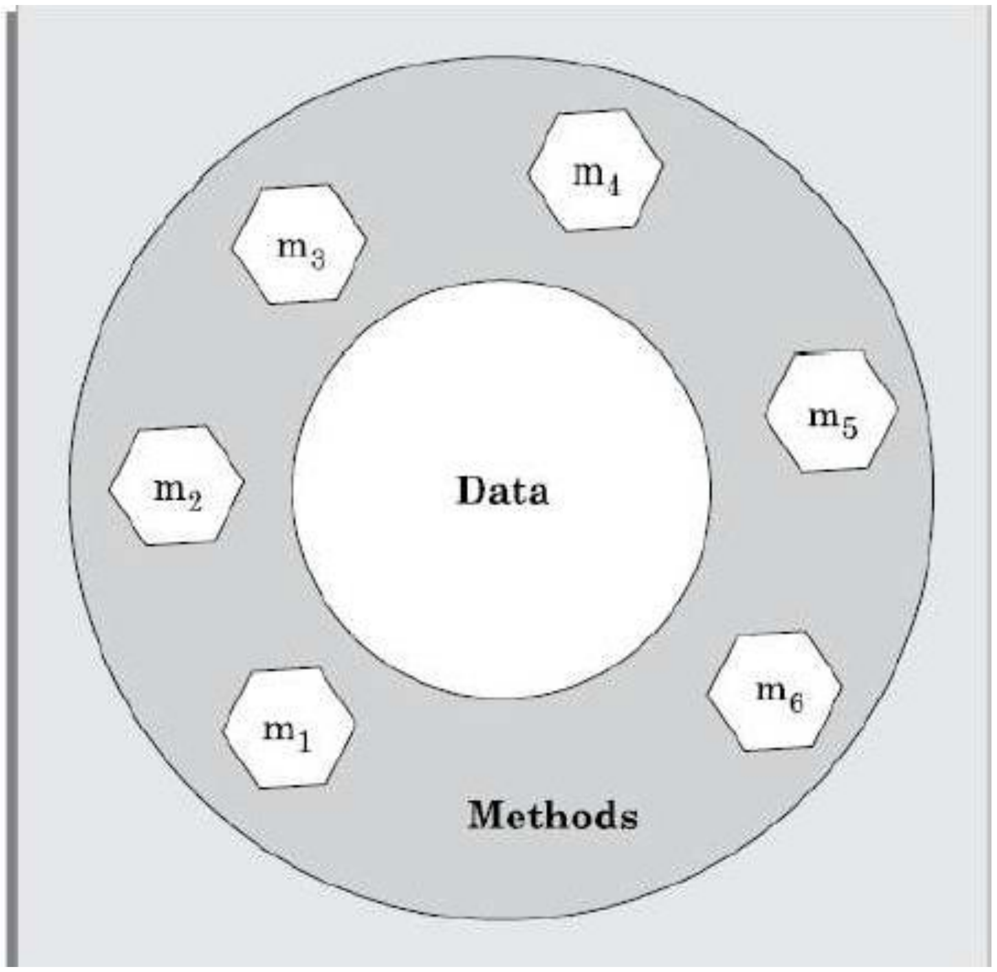
- The wrapping up of data(variables) and function (methods) into a single unit (called class) is known as encapsulation.


data
methods

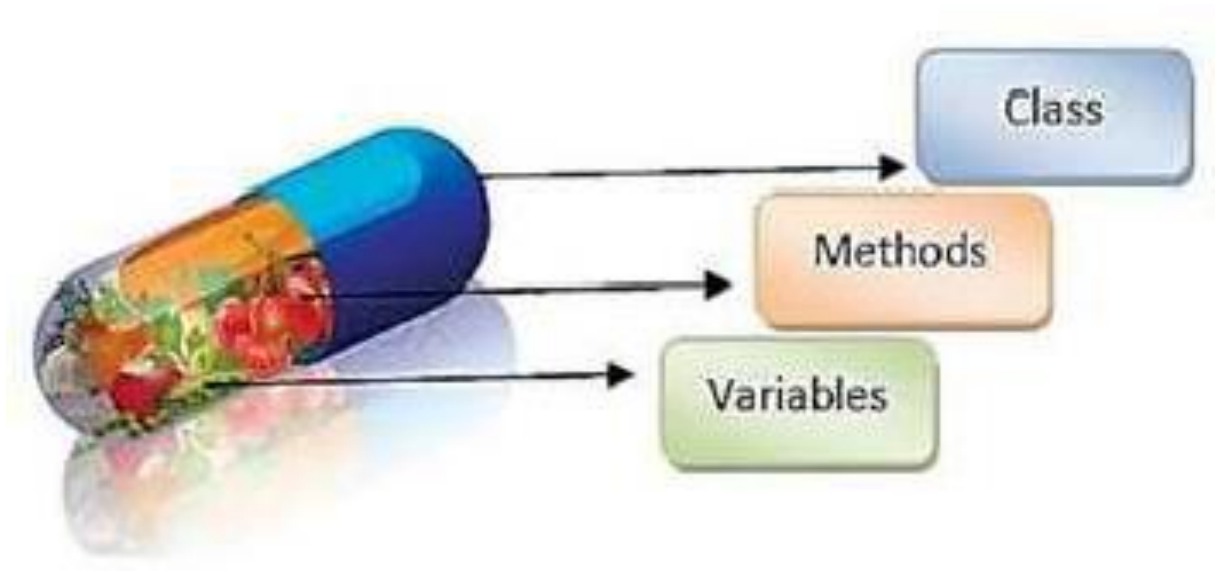
Encapsulation



Encapsulation



Encapsulation



Encapsulation - Advantage

Encapsulation offers the following three important advantages:

- **Protection from unauthorised data access:**
  - Protect data from accidental corruption and concurrent access(simultaneous access) problems.

  Data hiding-Helps to hide the internal structure data of an object.
  - provides abstraction, easier maintenance and bug correction.
- **Weak coupling-** Since objects do not directly change each others internal data, they are weakly coupled. This enhances understandability of the design.

Abstraction

- Abstraction mechanism
  - **consider only those aspects of the problem that are relevant** to a given purpose
  - and to suppress all aspects of the problem that are not relevant.
- Abstraction means displaying only essential information and **hiding** the inner details.

Abstraction(contd.)

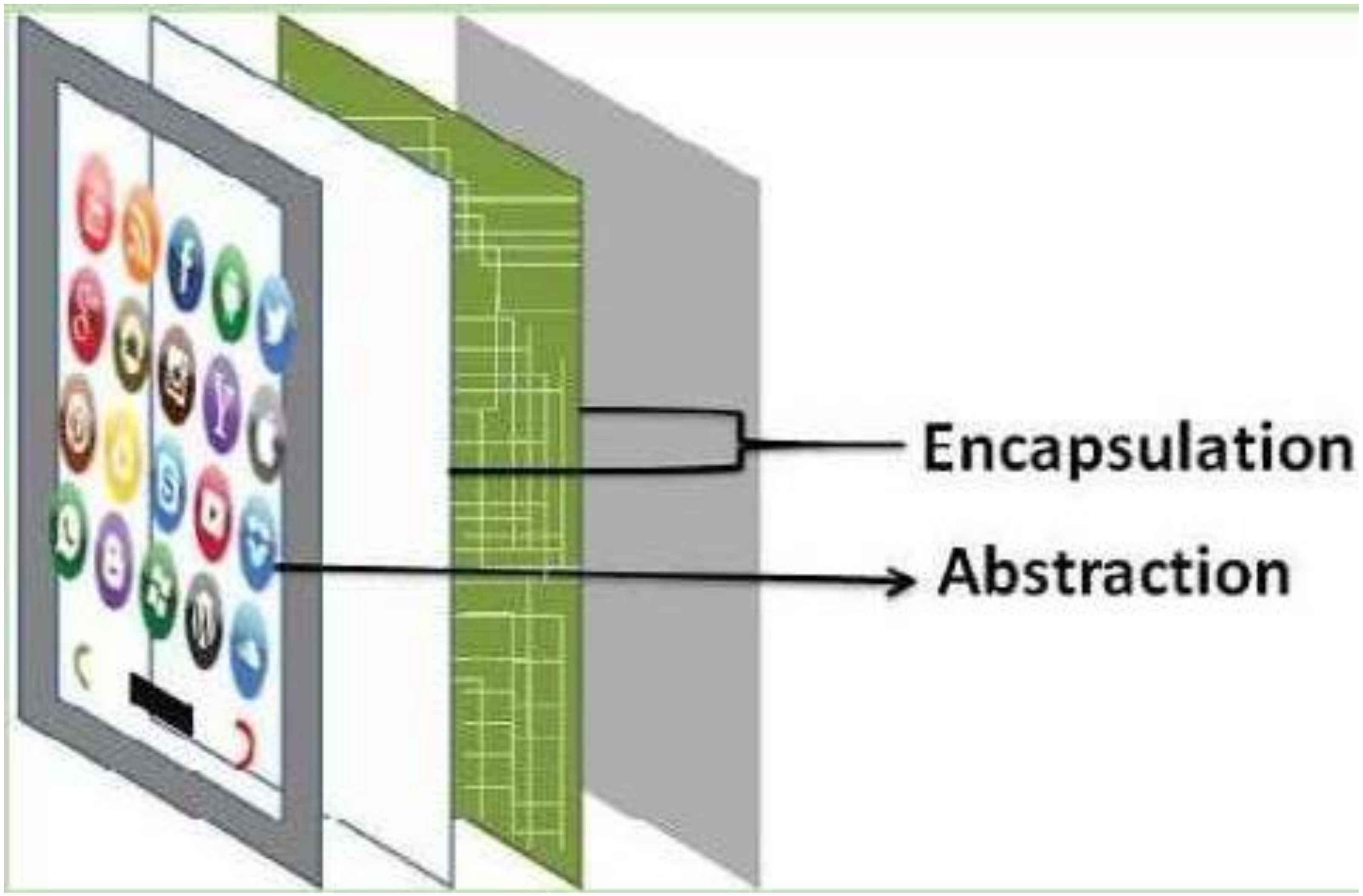
- Abstraction is supported in two different ways in an object-oriented designs (OODs).
  - **Feature abstraction-** A class hierarchy can be viewed as defining several levels (hierarchy) of abstraction, where each class is an abstraction of its subclasses.- Inheritance provides this.
  - **Data abstraction -** each object **hides** the exact way in which it **stores its internal information** from other objects .

Data abstraction-Real world example

- Consider a man driving a car. ***The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car***
- But he ***does not know about how*** on pressing accelerator the speed is actually increasing,
- *He does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.*
- This is what abstraction is.

Data abstraction -Advantage

- An important advantage of the principle of data abstraction is that
  - It reduces coupling among various objects
    - Objects do not directly access any data belonging to each other.
  - It leads to a reduction of the overall complexity of a design.
  - It helps in easy maintenance and code reuse.


Encapsulation
Abstraction

Class Relationships

Classes in a programming solution can be related to each other in the following four ways:

- Inheritance
- Association and link
- Aggregation and composition

Inheritance

- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- Inheritance is the process by which objects of one class acquired the properties of objects of another classes

Inheritance(contd.)

- **Derived Class**: The class that inherits properties from another class is called **Subclass** or Derived Class or **child** class.
- **Super Class** : The class whose properties are inherited by subclass is called **Base Class** Superclass or **parent**class.

  E.g. Doctor is a superclass. Surgeon and Neurologist are its subclasses

Inheritance

- A base class is said to be a **generalisation** of its derived classes.
- A base class is **specialized** into derived classes.
- This means that the base class contains properties (i.e., data and methods) that are common to all its derived classes.
- Derived class inherit the properties of base class. Derived can have their special own properties also.

Base class- Derived class

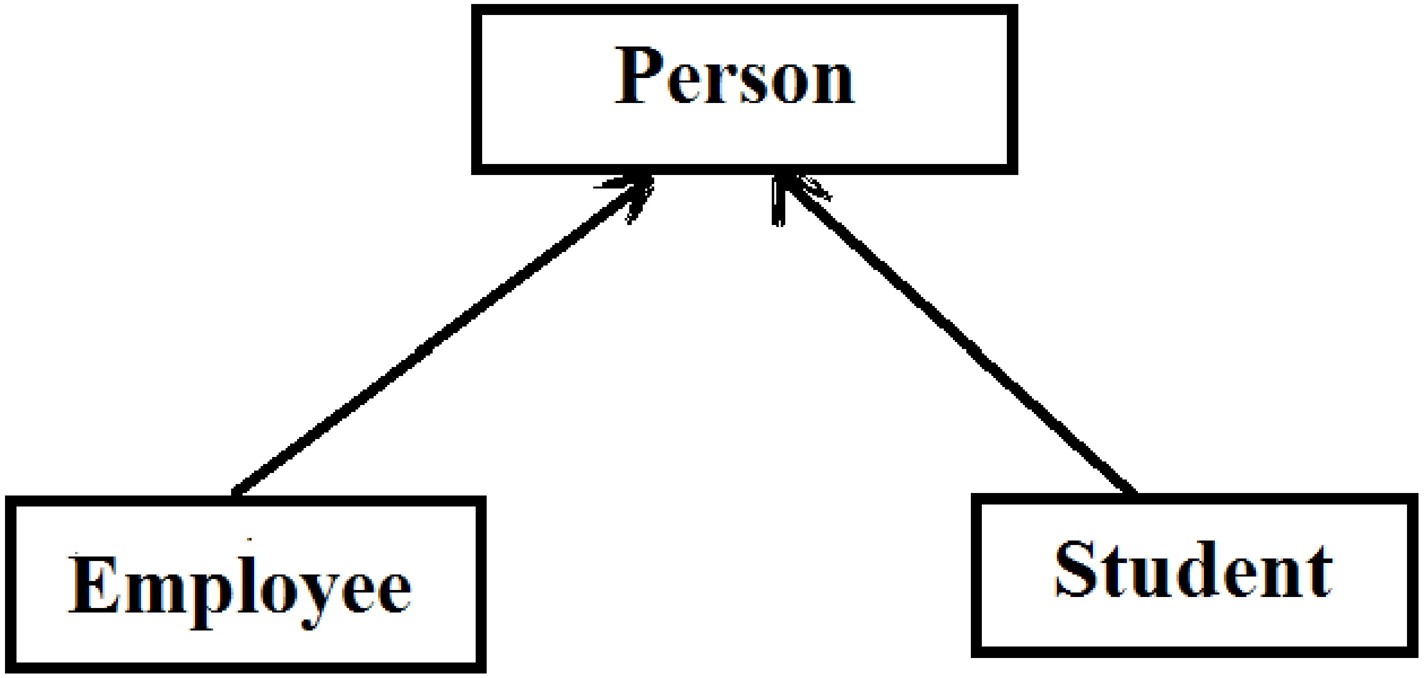
- Each derived class can be considered as a specialisation of its base class
  - because it modifies or extends the basic properties of the base class in certain ways.
- Therefore, the inheritance relationship can be viewed as a **generalisation-specialisation** relationship.

Inheritance - Example

- Consider the classes Person, Employee, Student.
  - Employee is a person
  - Student is a person
  - Employee and Student inherit the properties of Person class

Inheritance - Example

- Consider the classes Person, Employee, Student.
  - Employee is a person
  - Student is a person
  - Employee and Student inherit the properties of Person class
  - **Super(base) class** - Person
  - **Derived**(sub) class - Employee ,Student


Person
Employee
Student

Inheritance example

- Suppose Person class stores
  - ***data*** - name, aadhar number, address and data-of-birth
  - ***Methods***-enter_details(), modify_details().
- Employee is a subclass of class Person.
  - So Employee class *inherits all data and methods of Person class.*
  - It can also contain data and methods specific to employee such as also empid, designation, calculate_experience()

Inheritance example

- Suppose Person class stores
  - ***data*** - name, aadhar number, address and data-of-birth
  - ***Methods***-enter_details(), modify_details().
- Employee is a subclass of class Person.
  - contain data and methods specific to employee such as also empid, designation, calculate_experience()

Employee-

  - ***data*** - name, aadhar number, address and data-of-birth
  - ***Methods***-enter_details(), modify_details().

Inheritance - Example

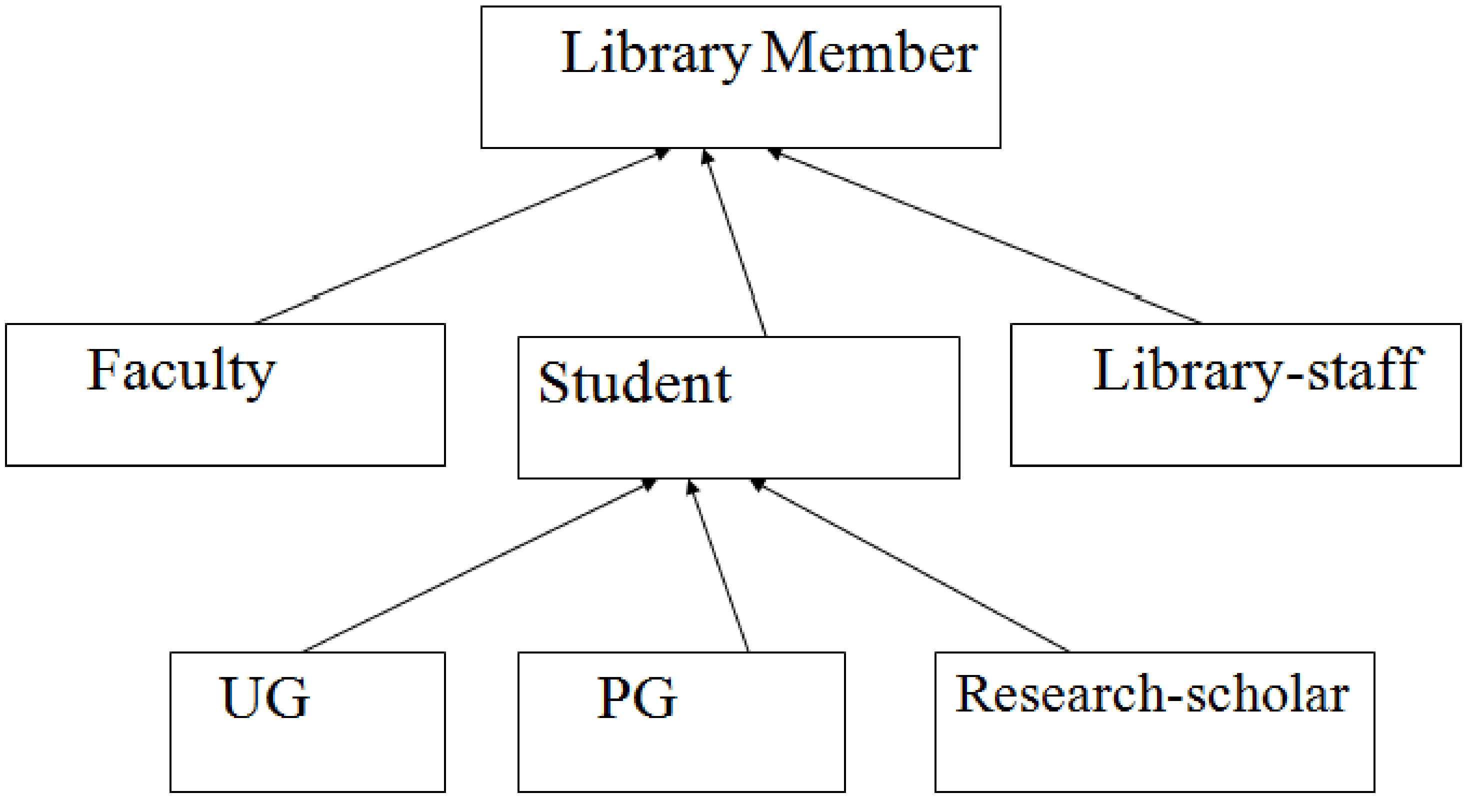
- Faculty, students, and library staff are library members.
- Base class(superclass)- ?
- Derived class(subclass) -?

Inheritance - Example

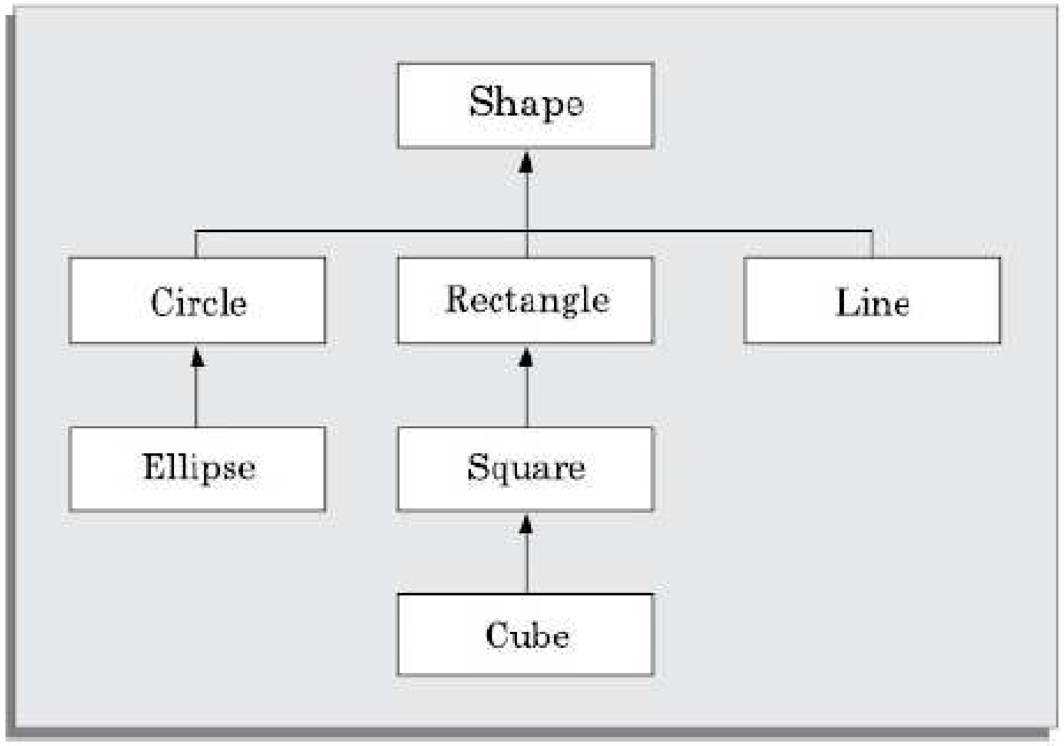
- Faculty, students, and library staff are library members.
- Base class(superclass)- Library member
- Derived class(subclass) -Faculty ,students , library staff

Inheritance - Example

- Faculty, students, and library staff are library members. Students fall in three categories PG, UG and research-scholars.



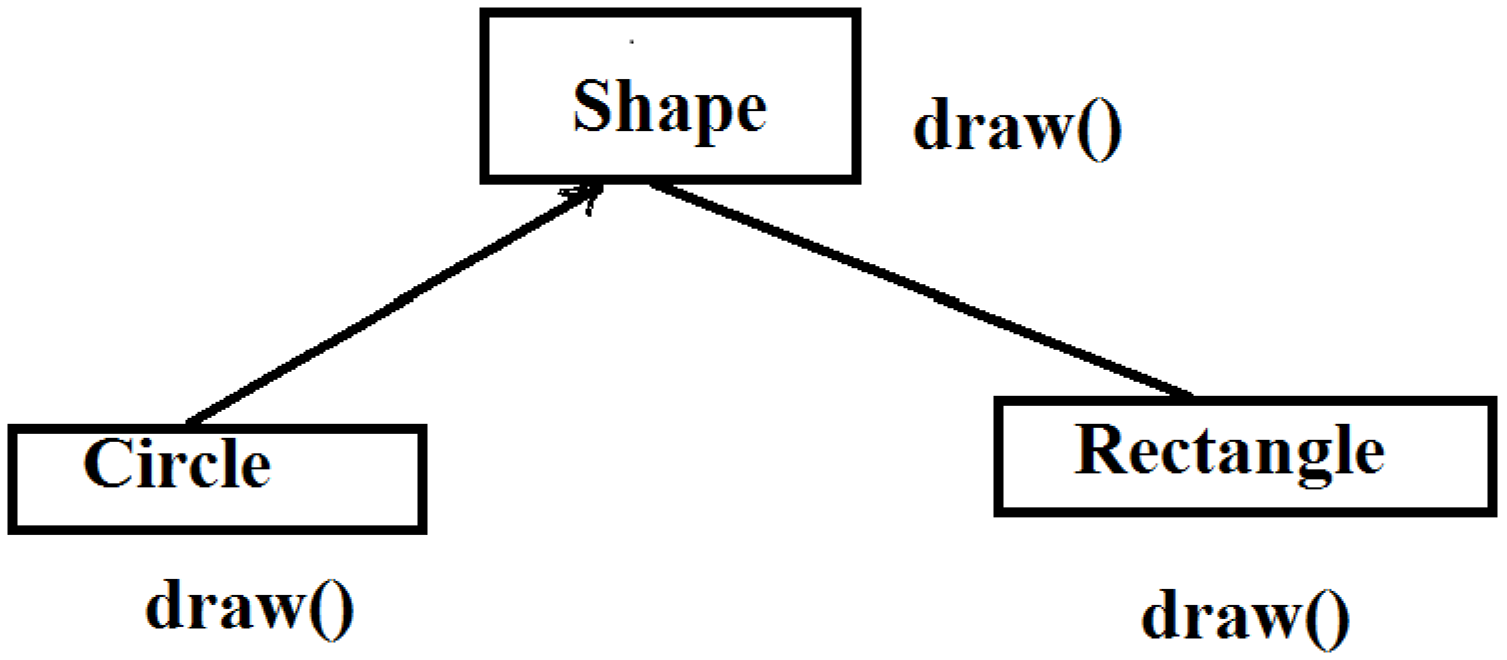
Class hierarchy



Class hierarchy of geometric objects

Method overriding

- When a method in the base class is also defined in a derived class, then the method is said to be overridden in the derived class.
- If subclass contain same method name superclass then subclass as method superclass method overrides

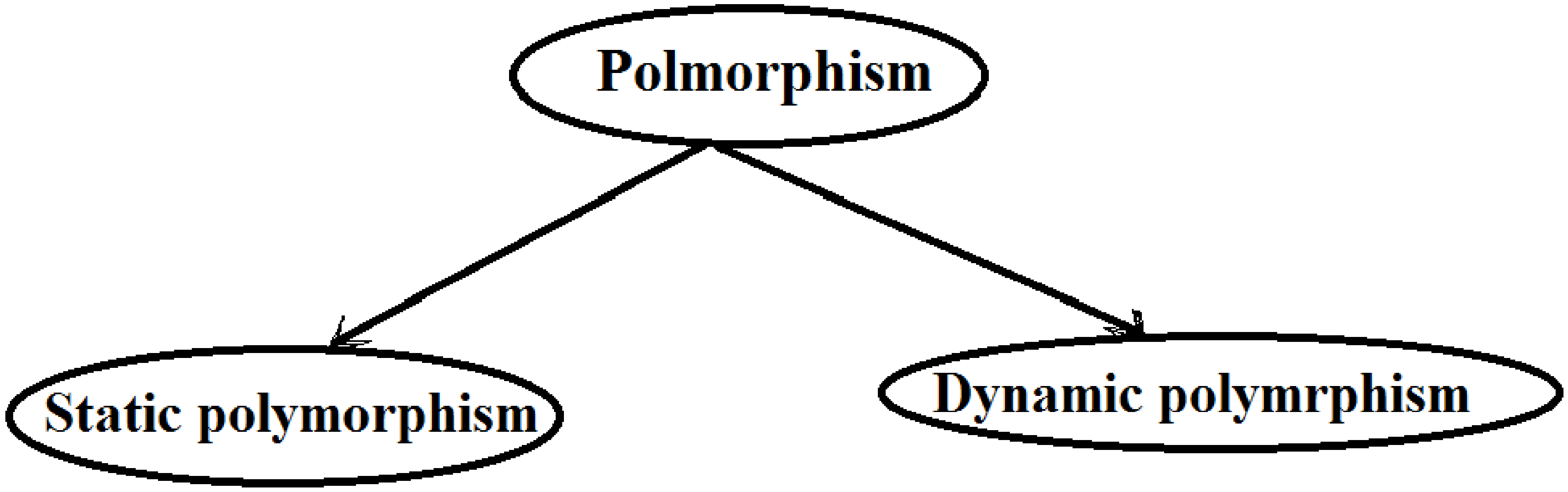


Method overriding

Method overriding

Polymorphism

- Polymorphism literally means poly (many) morphism (forms).
- Real world example
  - Diamond, graphite, and coal are called polymorphic forms of carbon


Polmorphism
Static polymorphism
Dynamic polymrphism

Static polymorphism

- Static polymorphism occurs when multiple methods implement the same operation.
- Static polymorphism is also called **static binding**.
- If a program has *different methods* with ***same name*** *but different parameter types*
  - when that method is invoked(called), (there are different methods with same name here), the exact method which is to be bound to the method-call is determined at ***compile-time*** (statically).

Method overloading- static binding

- **Method overloading** is a static polymorphism. Here a program (class) can have ***different functions with same name*** (but has different parameters).

Example-static polymorphism

Suppose class **Circle** has so many data and methods. Let Circle class has <u>three methods named *create*</u>

```
int create()

int create(int radius)

int create(int x, int y, int radius)
```

This is an example for **method overloading.**

Example-static polymorphism(contd.)

- ✓When *create function is called without parameter* then **int create()** method is invoked.
- ✓When *create function is called with one integer parameter* then **int create(**int radius**)** method is invoked.
- ✓When *create function is called with three integer parameters* then **int create**(int x, int y, int radius) method is invoked.

Dynamic polymorphism

- Dynamic polymorphism is also called **dynamic binding**.
- In dynamic binding, when a method is called , the exact method to be invoked (bound) is known at the **run time (dynamically)** and *cannot be determined at compile time.*

Dynamic polymorphism(contd.)

- Dynamic binding is based on two important concepts:
  - Assignment of an object to another compatible object.
  - Method overriding in a class hierarchy.

Dynamic polymorphism(contd.)

- **Assignment to compatible of objects**
  - In object-orientation, ***objects of the derived classes are compatible with the objects of the base class***.
  - That is, an object of the derived class can be assigned to an object of the base class, but not vice versa.
- **Method overriding**
  - If subclass contain same method name as superclass then subclass method overrides superclass method

Dynamic binding summary

- Even when the method of an object of the base class is invoked,
  - an appropriate overridden method of a derived class would be invoked
    - depending on the exact object that may have been assigned at the run-time to the object of the base class.

Advantage of dynamic binding

- The principal advantage of dynamic binding is that
  - it leads to **elegant programming** and facilitates **code reuse and maintenance**.
  - code is much more concise, understandable, and intellectually appealing

Relatioship between Objects

- Association and Links
- Aggregation
- Composition
- Dependancy

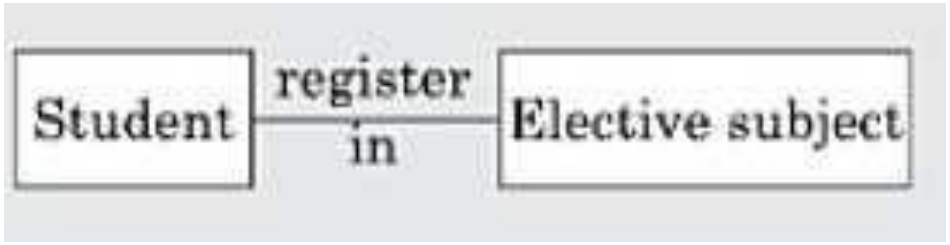
Association

- Association is a common type of relation among classes.
- The association relationship can either be ***bidirectional*** or ***unidirectional***.
- An association describes a ***group of similar links.***
  - A link can be considered as an instance of an association relation.
- An association between two classes simply means that *zero or more links may be present among the objects of the associated classes at any time during execution.*

Association

- When two classes are associated, they can take each others help (i.e. invoke each others methods) to serve user requests.(**binary association**)
  - if one class is associated with another *bidirectionally*, then the corresponding objects of the two classes know each others ids(identities).

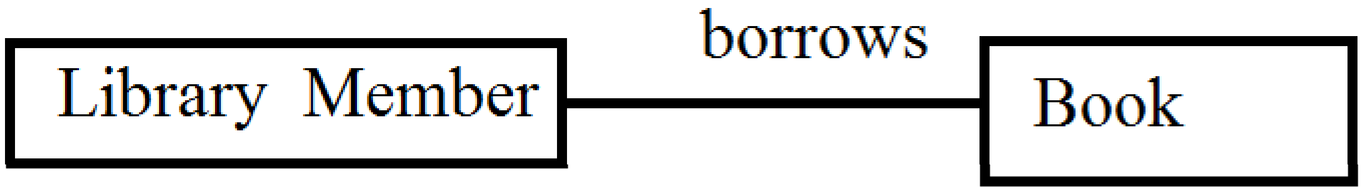
Association-Example



- A Student can register in one Elective subject.
  - Here, the class Student is associated with the class ElectiveSubject.
  - Therefore, an ElectiveSubject object would
    - *know the ids* of all Student objects that have registered for the Subject
    - and *can invoke their methods* such as printName, printRoll a nd enterGrade.
- This is example for binary association

Association-Example

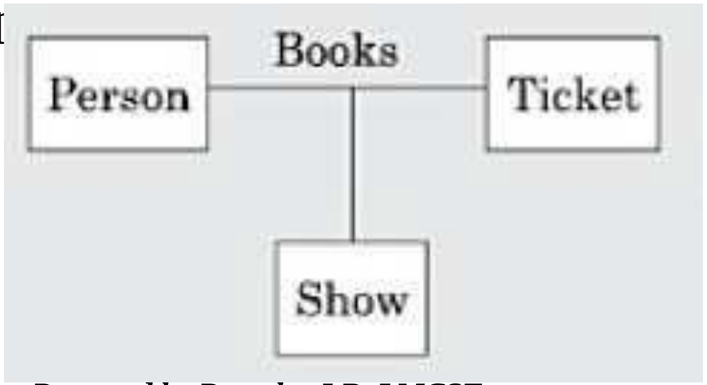
- Consider another example of association between two classes: Library Member borrows Books.



  – Here, ***borrows*** is the association between the class **LibraryMember** and the class **Book**.
  – The association relation would imply that given a book, it would be possible to determine the borrower and vice versa.

Association(contd.)

- **n-ary association**
  - Three or more different classes can be involved in an association.
  - If three classes are associated then it is called 3-ary (ternary) association
  - E.g. A person books a ticket for a certain show.
    - Here, an association exists among the Person, Ticket, and Show. This classes is associatioi tern ary

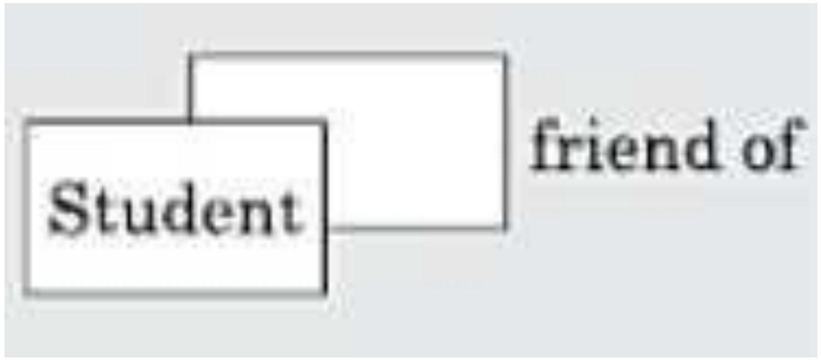


Association(contd.)

- A class can have an ***association relationship with itself.***

This is called

  - **recursive association** or **unary association.**
  - Example, consider the following—two students may be friends. Here, an association named *friendship* exists among pairs of objects of the Student class



Association(contd.)

- Links are time varying (or **dynamic**) in nature.
- Association relationship between two classes is **static** in nature.
  - If two classes are associated, then the association relationship exists at all points of time.
  - But **links between objects are dynamic in nature**.
  - Links between the objects of the associated classes can get formed and dissolved as the program executes.

Association(contd.)

- Example, an association relationship named **works-for** exists between the classes **Person** and **Company**.
  - Ram works for Infosys,
    - This implies that a link exists between the object Ram and the object Infosys.

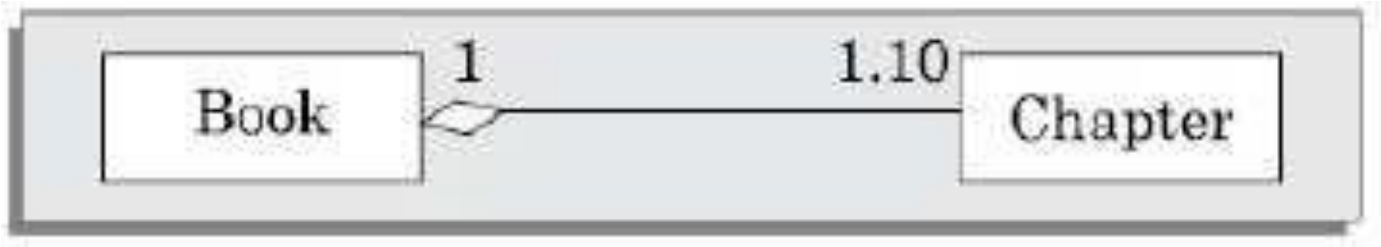
    Hari works for TCS
    - A works for link exists between the objects Hari and TCS.
  - If Ram may resign from Infosys and join Wipro. In this case, the link between Ram and Infosys breaks and a link between Ram and Wipro gets formed.
  - In all these case association works for remains there

Composition and aggregation

- Composition and aggregation represent **part/whole relationships** among objects.
- Composition/aggregation relationship is also known as **has a** relationship.
- Objects which contain other objects are called composite objects.

Composition and aggregation- Example

- Example: A Book object can have upto ten Chapters.
  - Here a Book object is said to be composed of upto ten Chapter objects.
  - A Book **has** upto ten Chapter objects



Composition and aggregation(contd.)

- Aggregation/composition can occur in a hierarchy of levels.
  - That is, an object may contain another object. This latter object may itself contain some other objects.
- Composition and aggregation relationships **cannot be reflexive.**
  - That is, an object cannot contain an object of the same type as itself.

Association and aggregation(contd.)

Association

Association in terms of objects refers to **"has a"** relationship between two related objects. For example, a employee has a communication address.

```
class Employee {
   String name;
   Address communicationAddress;
}
class Address {
   String address;
}
```

Association and aggregation(contd.)

Aggregation

Aggregation in terms of objects refers to "**has a"+** relationship between two related objects. For example, a department has multiple employees. It refers to having a collection of child objects in parent class. For example:

```
class Department {
  String name;
  List<Employee> employees;
}
class Employee {
  String name;
}
```

Composition and aggregation(contd.)

Composition

The composition is the strong type of association (“has a” or “has a+” relation. An association is said to composition if an Object owns another object and another object cannot exist without the owner object.

Composition and aggregation(contd.)

```
class Department {
   String name;
   List<Employee> employees;
   Public void addEmployee(String ename) {
   Employee e = new Employee(ename);
   employees.add(e);
   }
}
class Employee {
   String name;  Employee(String name) { this.name =
name;}
}
```

Dependency

- A class is said to be dependent on another class,
  - if any changes to the latter class requires a change to be made to the dependent class.

  E.g. class1 is dependent on class2 if any change is made in class2 then change is required in class1 too.
- A dependency relation between two classes shows that **any change made to the independent class** would require the *corresponding change to be made to the dependent class*.

Dependency(contd.)

- Two important reasons for dependency to exist between two classes are the following:
  - A ***method*** of a class takes an **object of another class as an argument**.
    - E.g. class A

      {

      int function(class B){ ……………..}

      }
  - A **class** *implements* an **interface** class.
    - If some properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.

Summary of class relationship

- **Aggregation**
  - B is a part of A
  - A contains B
  - A is a collection of Bs
- **Composition**
- B is a permanent part of A
- A is made up of Bs
- A is a permanent collection of Bs

Summary of class relationship

- **Inheritance**
  - A is a kind of B
  - A is a specialisation of B
  - A behaves like B

  Association
  - A delegates to B
  - A needs help from B
  - A collaborates with B. Here collaborates with can be any of a large variety of collaborations that are possible among classes such as employs, credits, precedes, succeeds, teaches etc.

Abstract class

instances(objects)

- Classes that are not intended to produce instances (Objects) are called abstract casses

are

– an abstract class cannot be instantiated.

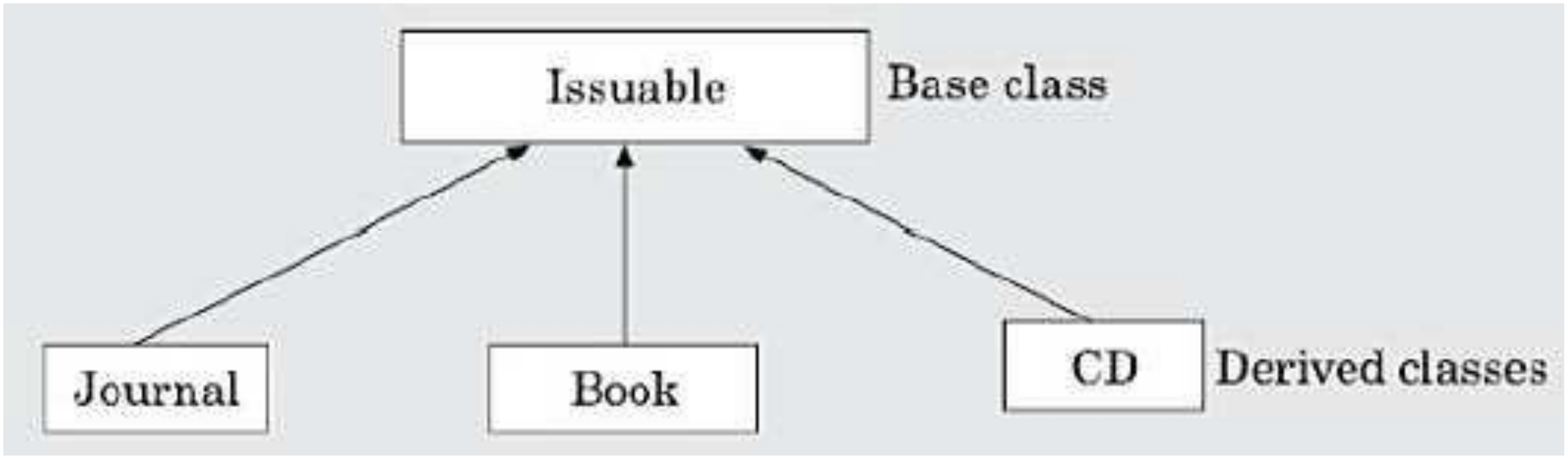
Objects cannot be created from abstract class.

– Abstract class ***act as base class*** in inheritance.

• Abstract class usually support generic methods.

- Advantage
  - code reuse can be enhanced
  - the effort required to develop software brought down.

Abstract class- Example



- Here Issuable is an abstract class and cannot be instantiated.

Advantages of OOD

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design undestandability enabling development of large programs

Disadvantages of OOD

- The principles of abstraction, data hiding, inheritance, etc. do incur ***runtime overhead***.
- An important consequence of object-orientation is that the data that is centralized in a procedural implementation, gets ***scattered across various objects*** in an object-oriented implementation.

Object-oriented Programming Language(OOPL)

- The first object-oriented programming language was Smalltalk.
- Other OOPL are C++, Java etc.

Reference Text Book

- Rajib Mall, Fundamentals of Software Engineering, 4th edition, PHI, 2014.

Thank you

INTRODUCTION

- JAVA (Part 1)

(Module 1)-

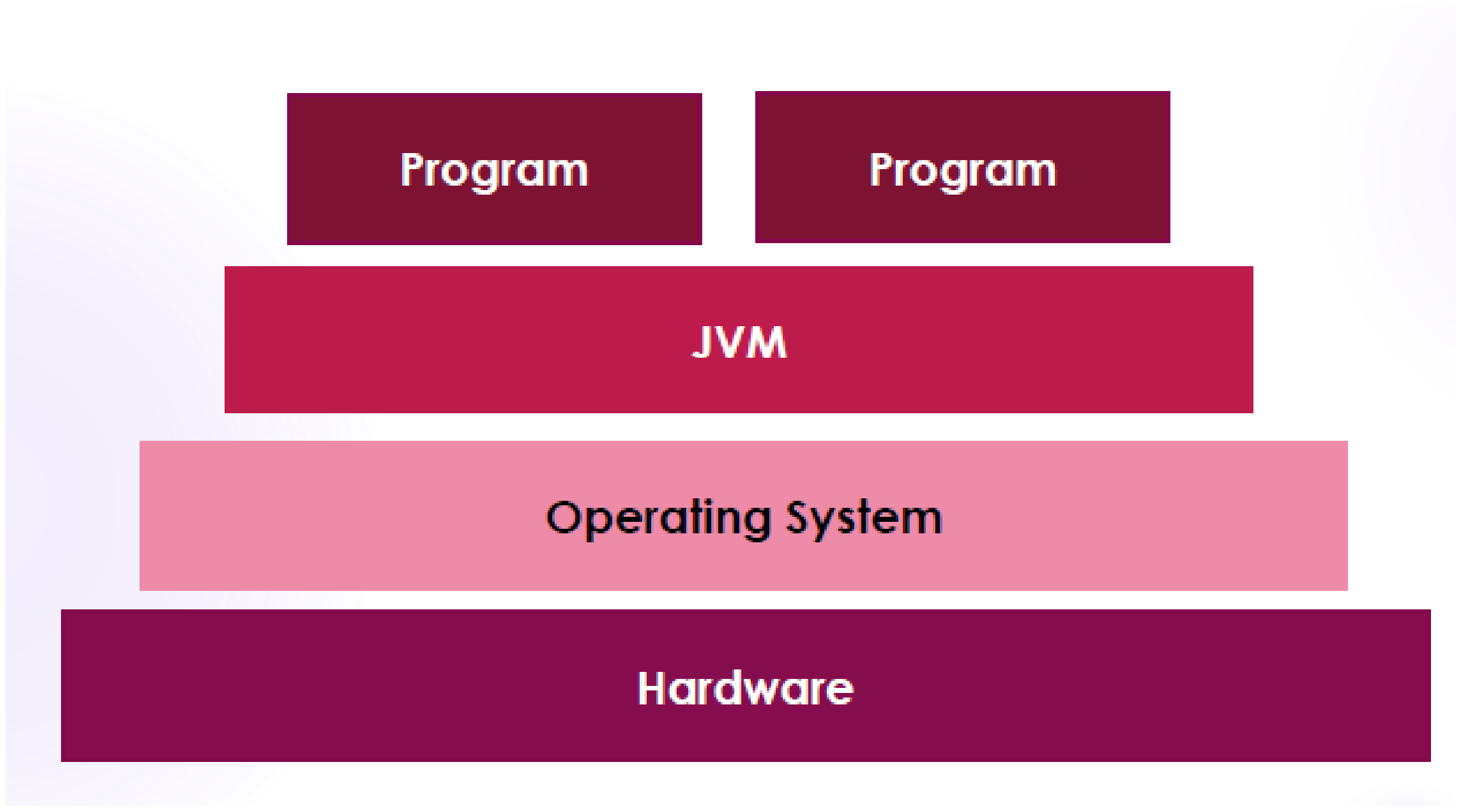
Prepared by Renetha J.B.(LMCST)

Topics

- ✓ Java programming Environment and Runtime Environment,
- ✓ Development Platforms
  - ➢ Standard, Enterprise.
- ✓ JVM
- ✓ Java compiler,
- ✓ Bytecode

Java

- Developed by **James Gosling** from **Sun Microsystems** in 1991.
  - This language was initially called "**Oak**," but was renamed "Java" in 1995.
- The target of Java is to **write a program once and then run this program on multiple operating systems**. (**WORA**)
- Java is a programming language
  - It has compiler, core libraries and a runtime (Java virtual machine(JVM)).

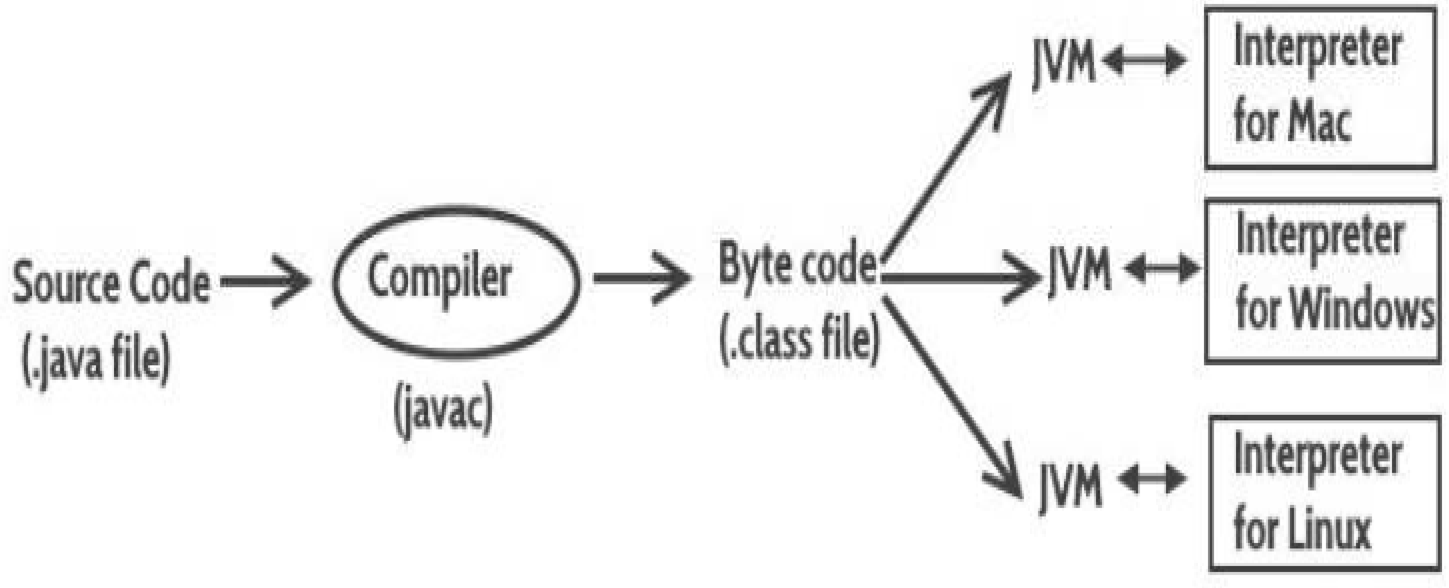

Program
Program
JVM
Operating System
Hardware

JVM

- JVM is the Java run-time system.
- **Java Virtual Machine** is called virtual because it **provides a machine interface** that *does not depend on* the operating system and machine hardware architecture.
- So Java programs are WORA (Write Once Run Anywhere)p rograms.

JVM(contd.)

- When we compile a Java program, we get **.class** file(bytecode) which is not executable.
- JVM **interprets the .class file into machine code** depending on the operating system and hardware.
- JVM **executes java programs like a machine.**
- JVM is also responsible for **garbage collection, array bond checking etc.**
- JVM is platform indepenedent.


Source Code (.java file)
Compiler
(javac)
Byte code (.class file)
JVM
Interpreter for Mac
JVM
Interpreter for Windows
JVM
Interpreter for Linux

JRE

- It is **an installation package** which provides ***environment to only run*** (not develop)the java program (or application) onto your machine.

- JRE is only used by END-USERS of the system who only wants to run the Java programs

- The JDK, along with the Java Virtual Machine (JVM) and the JRE, can be used by developers to program and run Java applications.

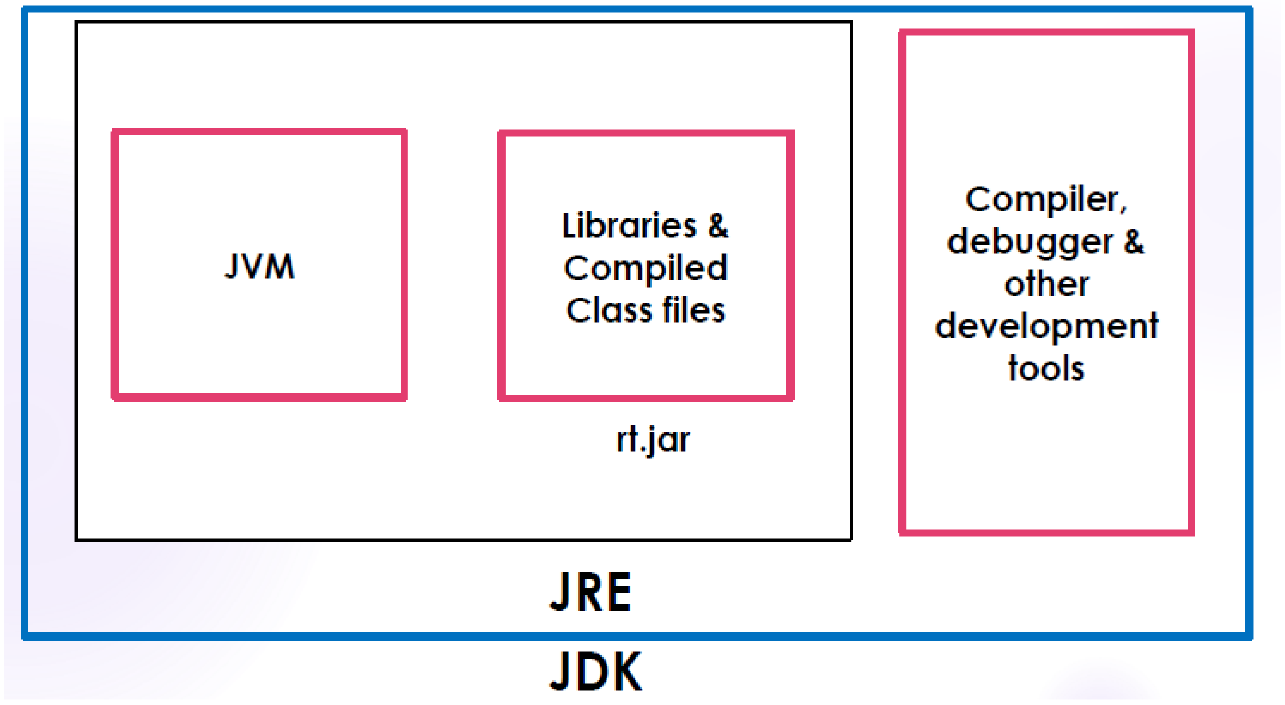
JDK

- The Java Development Kit (JDK) is a ***software development environment*** used for developing Java applications.
- **It includes**
  - The **Java Runtime Environment (JRE)**
  - An interpreter/loader (**Java**)
  - A compiler (**javac**)
  - An archive (**jar**)
  - A documentation generator (**Javadoc**)
  - Other tools needed in Java development.

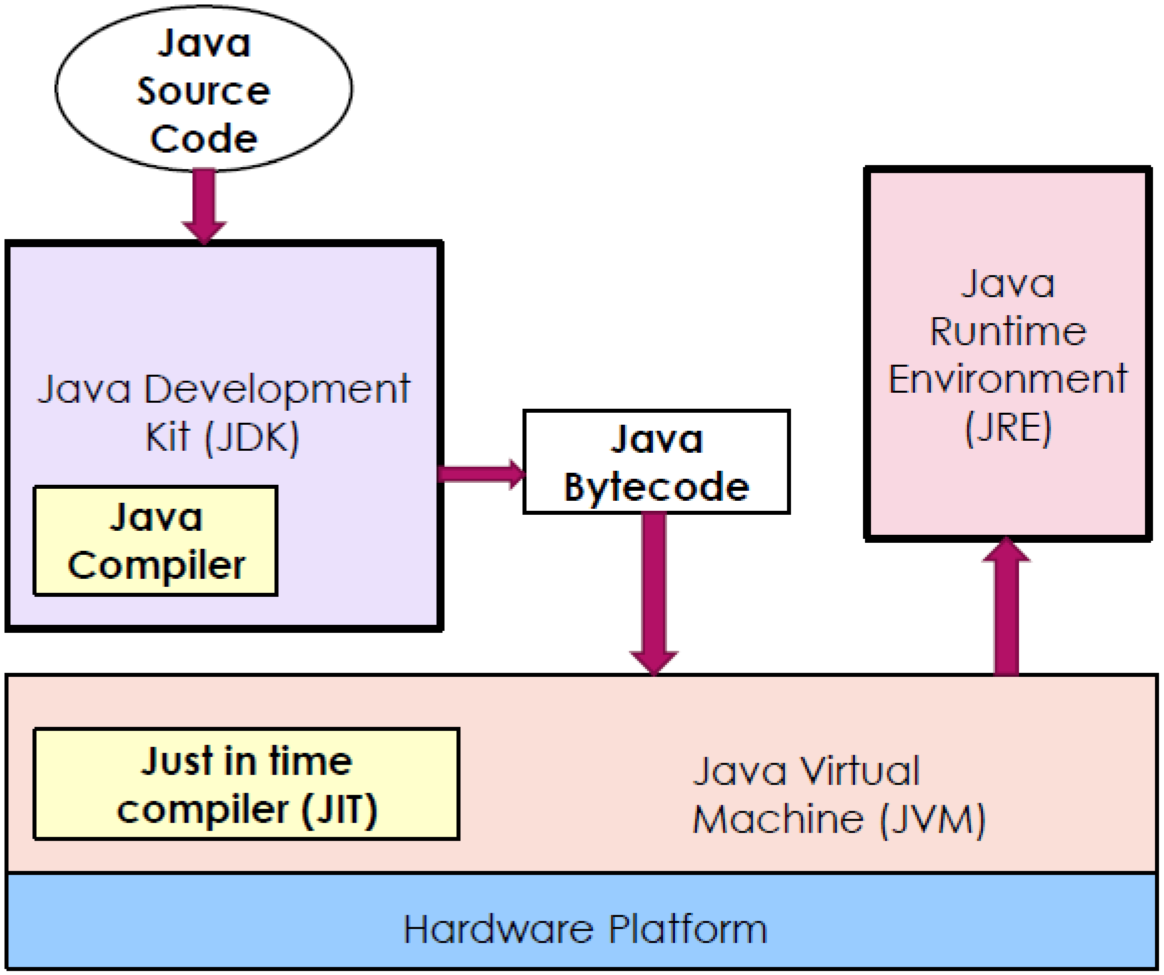
JDK

- JDK provides the **environment to develop and execute(run)**the Java program.
- JDK is a kit(or package) which includes two things
  - Development Tools(to provide an environment to develop your java programs)
  - JRE (to execute your java program).
- **JDK is used by Java Developers.**

Jvm jre jdk


JVM
Libraries & Compiled Class files
rt.jar
Compiler, debugger & other development tools
JRE
JDK

Interaction between JDK and JRE


Java Source Code
Java Development Kit (JDK)
Java Compiler
Java Bytecode
Java Runtime Environment (JRE)
Just in time compiler (JIT)
Java Virtual Machine (JVM)
Hardware Platform

Java Programming Environment

- Java is a **concurrent, class-based, object-oriented programming and runtime environment**, consisting of
  - A programming language
  - An API specification
  - A virtual machine specification

Development Platforms

- All Java platforms consist of a **Java Virtual Machine (JVM)** and an **Application Programming interface (API).**
  - The Java Virtual Machine is a program, for a particular hardware and software platform, that *runs Java technology applications*.
  - An API is a collection of software components that you can *use to create other software components or applications*.

Development Platforms

- Java development platform is a particular ***environment in which Java programming language applications run.***
  - Java Platform, Standard Edition (Java SE)
  - Java Platform, Enterprise Edition (Java EE)
  - Java Platform, Micro Edition (Java ME)
  - Java FX

Development Platforms - Standard Edition

- When most people think of the Java programming language, they think of the Java SE (**Standard Edition**) API.

- Java **SE**'s API **provides the core functionality of the Java** programming language.

- It **defines** everything from the **basic types and objects** of the Java programming language to **high-level classes** for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

- Java SE platform **consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits** commonly used in Java technology applications.

Development Platforms –Enterprise Edition

- The Java EE (**Enterprise Edition**) platform is built on top of the Java SE platform.

- The Java EE platform provides an API and runtime environment for developing and running **large-scale, multi-tiered, scalable, reliable, and secure network applications.**

Development Platforms –Micro Edition

- The Java ME platform provides an API and a small-footprint virtual machine for **running Java programming language applications on small devices, like mobile phones.**
- This API is a subset of the Java SE API, along with special class libraries useful for **small device application development**.

Development Platforms –Java FX

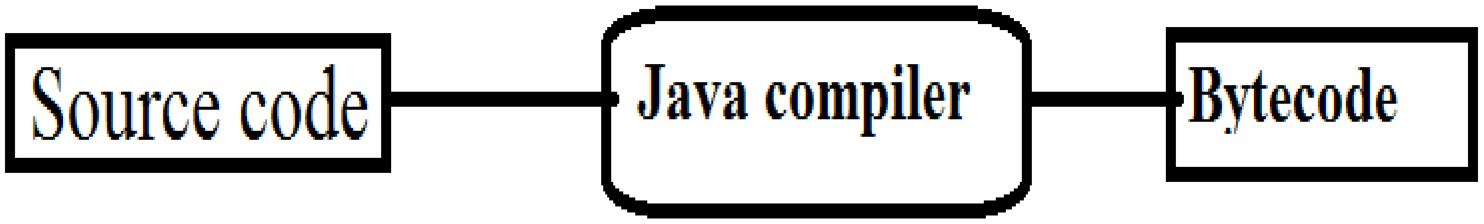
- Java FX technology is a platform for **creating rich internet applications written in Java FX Script**.
- Java FX Script is a statically-typed declarative language that is compiled to Java technology bytecode, which can then be run on a Java VM.

Java Compiler

- A Java compiler is a compiler for the Java programming language.
- Java programs are compiled using **javac** command.
- Command for compilation

  javac Programname***.java***

- The output of compiling the java code is not executable code. It is called **bytecode** (Programname***.class***)



Program.java **Program.class**

Java's Magic: The Bytecode

- The output of compiling the java code is not executable code. It is called **bytecode.**
- *Bytecode is* a highly optimized set of instructions designed **to be executed by the Java run-time system**, which is called the *Java Virtual Machine (JVM)*.
- JVM was designed as an ***interpreter*** *for bytecode*.
- Bytecode is a class file.

Bytecode(contd.)

- Translating a Java program into bytecode **makes it much easier to run a program in a wide variety of environments** because only the JVM needs to be implemented for each platform. -PORTABILITY
- Although the details of the JVM will differ from platform to platform, all JVM **understand the same Java bytecode.**
- Bytecode has been highly *optimized*, so the use of bytecode enables the JVM to **execute programs much faster**.(eventhough compilation and interpretation is needed)

Bytecode(contd.)

- When a **JIT(Just In Time) compiler** is part of the JVM, *selected portions of bytecode are compiled* into executable code in real time, on a piece-by-piece, demand basis.
- JIT compiler compiles code as it is needed, during execution.
  - Not all sequences of bytecode are compiled—only codes that will benefit from compilation.
  - The remaining code is simply interpreted.
- ***Java is a compiled interpreted language.***

Bytecode(contd.)

- Java bytecode is the **intermediate representation** of your Java program that **contains instructions that Java Virtual Machine will execute**.
- Thus, the output of javac is **not code that can be directly executed.**

REFERENCE

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

INTRODUCTION

- JAVA(Part 2)

Prepared by Renetha J.B.(LMCST)

Topics

- ✓ Java applet
- ✓ Java Buzzwords
- ✓ Java program structure
- ✓ Comments
- ✓ Garbage Collection
- ✓ Lexical Issues

Java applet

- An applet is a special kind of **small Java program** that is designed to be
  - **transmitted over the Internet** and
  - **automatically executed by a Java-compatible web browser.**
- If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.
- The **applets** are **used** to provide interactive features to web applications.

Java applet(contd.)

- Applets are typically used to
  - **display data** provided by the server,
  - **handle user input**, or
  - provide **simple functions**, such as a loan calculator, that **execute locally**, rather than on the server.
- Applet allows some functionality to be moved from the server to the client.
- Applet is a **dynamic, self-executing program.**

Applet - features

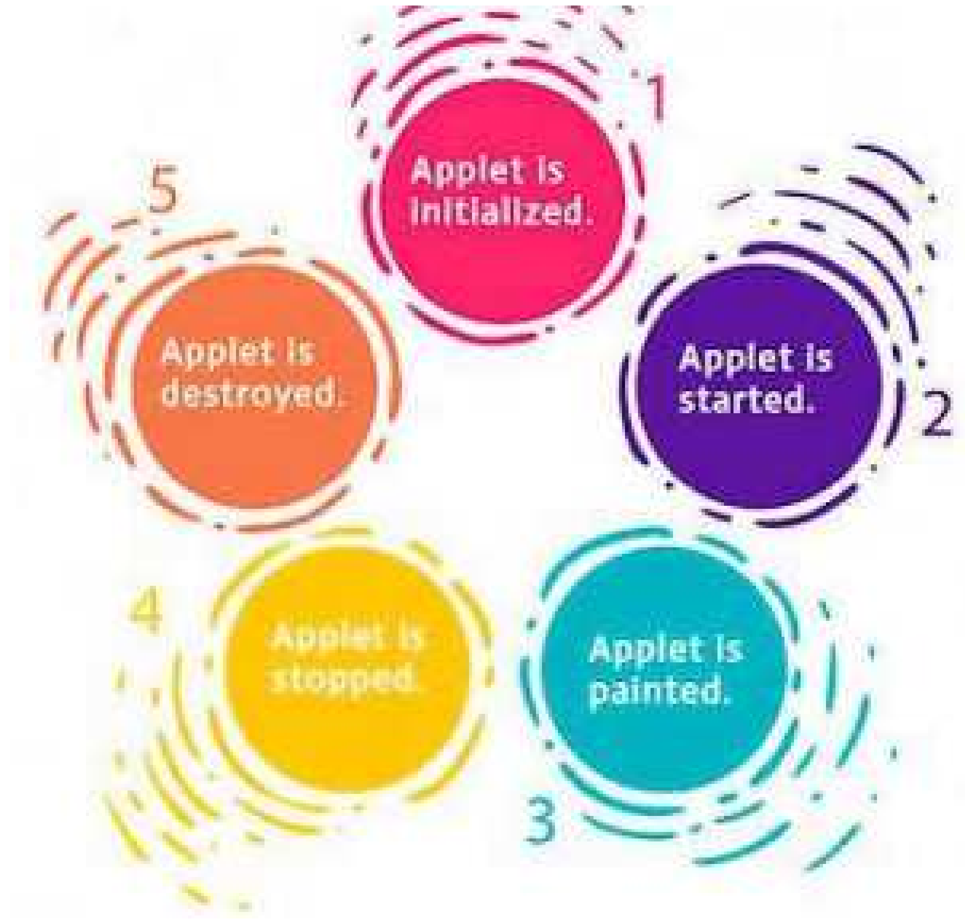
- Usually a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments.
- Java solved these problems in an effective and elegant way.
  - **Security-** Java achieved security by confining an applet to the Java execution environment and **not allowing it access to other parts of the computer.**
  - **Portability-** Same applet can be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.

Applet(contd.)

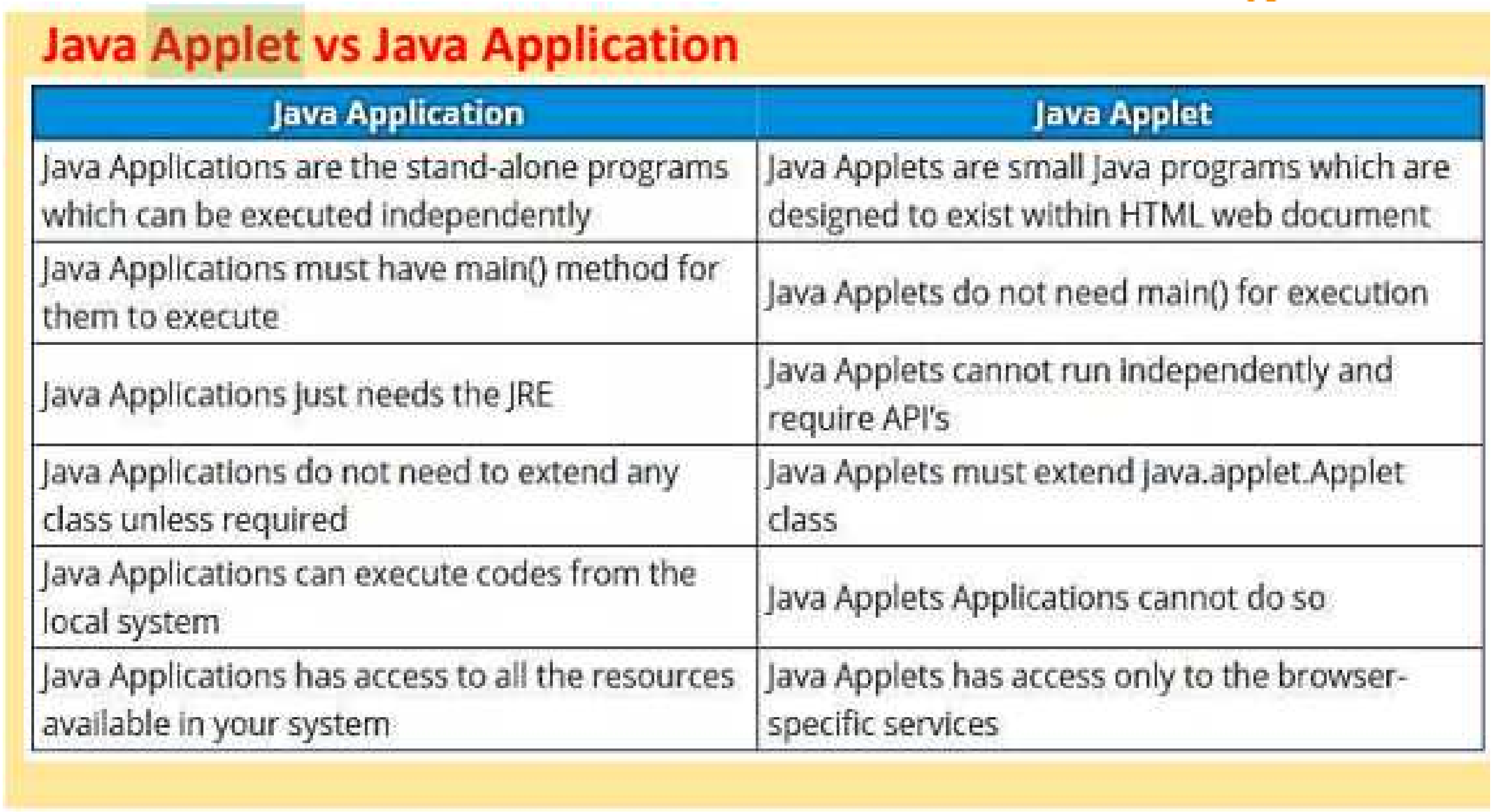
- Applets are
  - small applications
  - that are accessed on an Internet server,
  - transported over the Internet,
  - automatically installed,
  - and run as part of a web-document.

Applet(contd

- Applets are **not stand-alone programs**.
  - Instead, they run within either a web browser or an applet viewer.
- JDK provides a standard applet viewer tool called **applet viewer**.
- In general, **execution of an applet does not begin at main() method.**



Lifecycle of Java Applet



Java Applet vs Java Application

Java Application	Java Applet
Java Applications are the stand-alone programs which can be executed independently	Java Applets are small Java programs which are designed to exist within HTML web document
Java Applications must have main() method for them to execute	Java Applets do not need main() for execution
Java Applications just needs the JRE	Java Applets cannot run independently and require API's
Java Applications do not need to extend any class unless required	Java Applets must extend java.applet.Applet class
Java Applications can execute codes from the local system	Java Applets Applications cannot do so
Java Applications has access to all the resources available in your system	Java Applets has access only to the browser-specific services

Java Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Java Buzzwords(contd.)

- **Simple**
  - easy for the professional programmer to learn and use effectively.
- **Security**
  - Java achieved security by confining to the Java execution environment and not allowing it access to other parts of the computer.
- **Portable**
  - Same Java program can be executed by the wide variety of CPUs, operating systems etc.

Java Buzzwords(contd.)

- **Object-Oriented**
  - Atleast one class should be there.
  - everything is an object.
  - The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects

Java Buzzwords(contd.)

- **Robust**
  - the program **must execute reliably in a variety of system.**
  - Ability to create robust programs was given a high priority in the design of Java.
- To gain reliability,
  - Java restricts us in a few key areas to **force us to find mistakes** early in program development.
  - Java **frees us** from having to worry about many of the **most common causes of programming errors**.
  - Because Java is a strictly typed language, it **checks code at compile time**. However, it also **checks code at run time**.

Why Java is Robust?(contd.)

- Two main reasons for program failure are **memory management mistakes** and **mishandled exceptional conditions** (that is, run-time errors).
  - Memory management and Exception can be a difficult, tedious task in traditional programming environments.
- Java virtually eliminates these problems by **managing memory allocation and deallocation** for us.
- Java helps in handling exceptional conditions by **providing object-oriented exception handling.**
- **So Java is robust.**

Java Buzzwords(contd.)

- **Multithreaded**
  - Java was designed to meet the real-world requirement of creating interactive, networked programs.
  - To accomplish this, Java supports multithreaded programming, which allows to **write programs that do many things simultaneously.**
- **Architecture-Neutral**
  - A central issue for the Java designers was that of code longevity and portability.
  - Their goal was "write once; run anywhere, any time, forever." **WORA**

Java Buzzwords(contd.)

- **Interpreted and High Performance**
  - Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode, which can be executed by JVM.
  - Java bytecode can be easily **translated directly into native machine code** for **very high performance** **by using a just-in-time compiler.(JIT)**
  - Java run-time systems that provide this feature.

Java Buzzwords(contd.)

- **Distributed**
  - Java is designed for the distributed environment of the Internet because **it handles TCP/IP protocols.**
  - Java also supports ***Remote Method Invocation (RMI)*.**
    - *This feature enables a program to* invoke methods across a network.

Java Buzzwords(contd.)

- **Dynamic**
  - Java programs **carry** with them substantial amounts of **run-time type information** that is used to verify and resolve accesses to objects at run time.
  - This makes it possible to **dynamically link code** in a safe and expedient manner.
  - Small fragments of bytecode may be dynamically updated on a running system.

Java

- Java is related to C++, which is a direct descendant of C.
- From C, Java derives its syntax.
- Many of Java's object-oriented features were influenced by C++.

Java program

- Write and save the source file(program code) as **Example.java** in any directory.

  E.g. D:\oop

- **Compile** the program

  javac Example.java

  – If there is ***error,*** then correct and save and compile program again.

  – The javac compiler creates a file called **Example.class** that contains the bytecode version of the program.

Java program

- If there is **no error after compilation** you can **execute** the program using Java application launcher, called **java**

  java Example

  Here we are passing the class name *Example* as a command-line argument.
- i.e. When we execute **java command** we are actually specifying the name of the class that you want to execute.
  - It will automatically search for a file by that **name that has the .class extension.** ***(Here Example.class)***
  - **If it finds the file, it will execute the code** contained in the specified class.

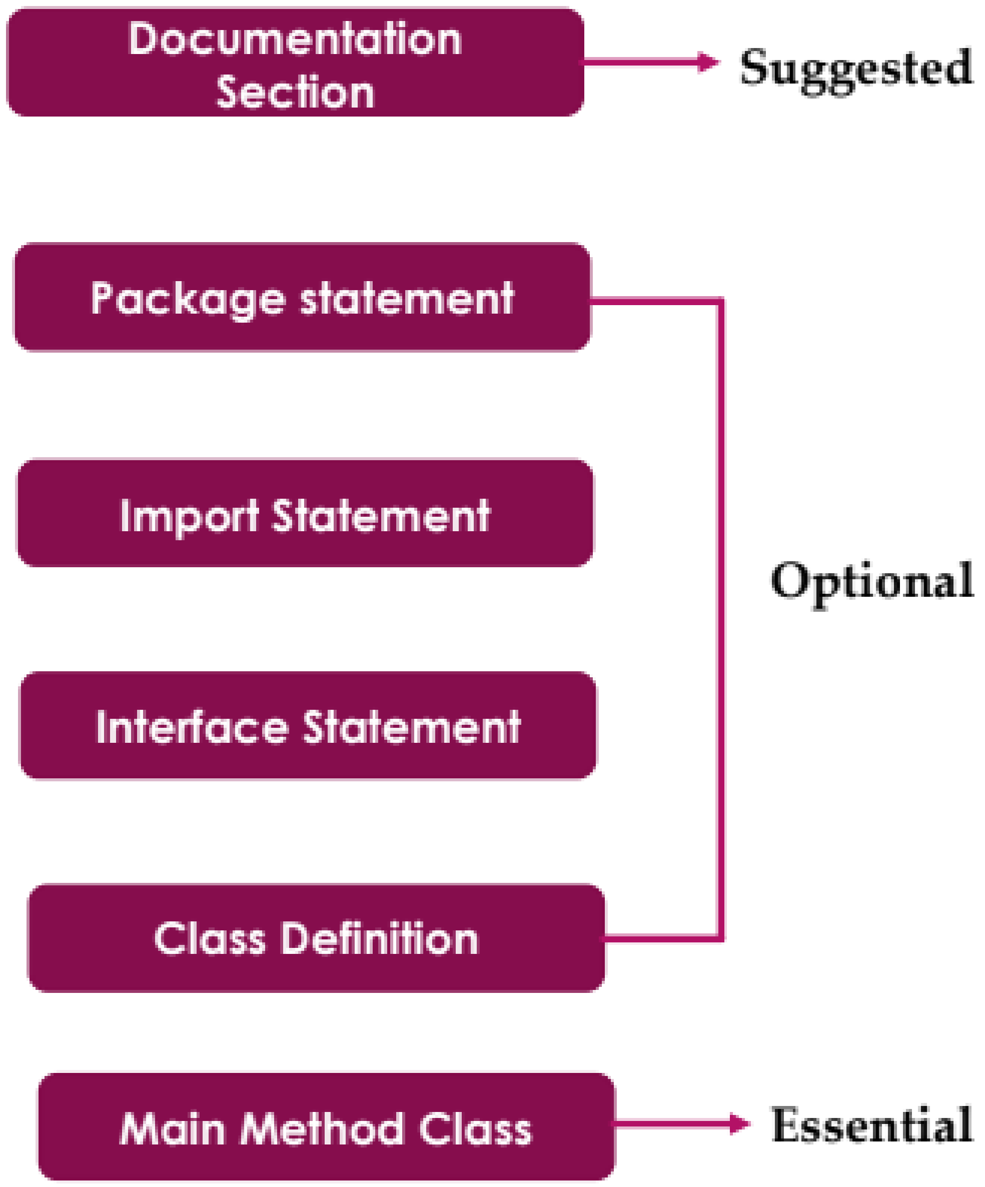
Java program

- In Java, a source file is officially called a *compilation unit*.
- *It is a file that contains one* or more class definitions.
- The Java compiler requires that a source file use the **.java** filename extension.
- In Java, **all code must reside inside a class.**
- By convention, **the name of that class in program should match the name of the file** that holds the program.
- Java programs are **case sensitive.**

Java program

- If there is **no public class** then **file name** may be same or different from the **class name** in **Java**.
- If there is a **public class** then the file name is same as the public class.

Java program structure



Documentation Section

- We can write comments in this section.
- Comments are beneficial for the programmer because they help them understand the code.
- There are three types of comments that Java supports
  – Single line Comment
  – Multi-line Comment
  – Documentation Comment

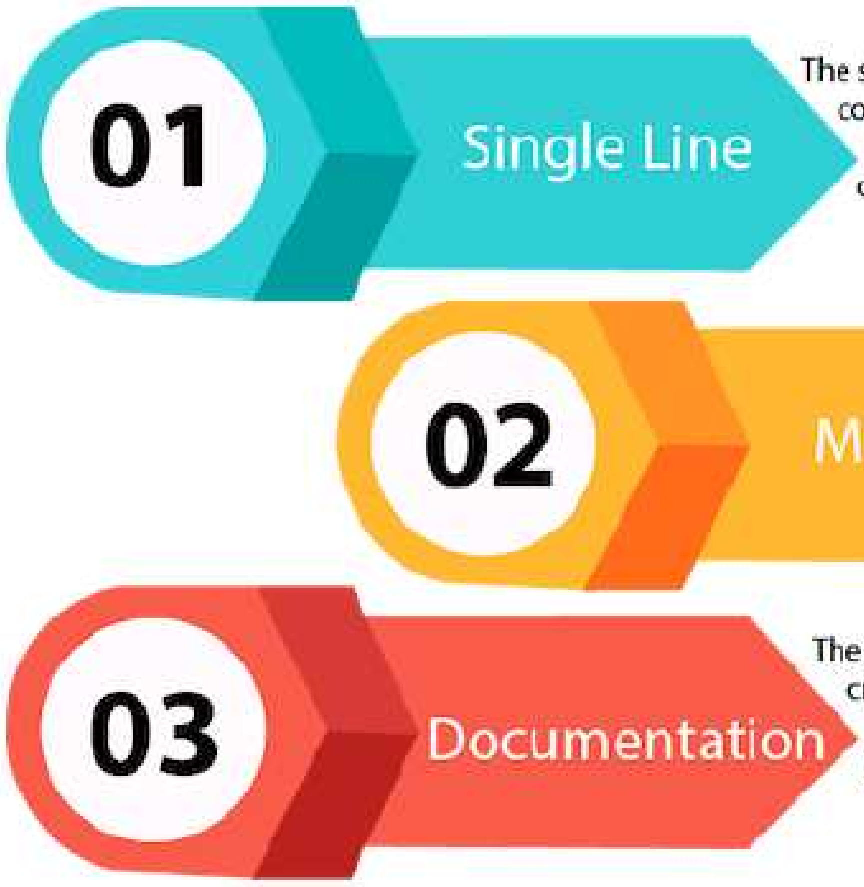
Comments

- Single line Comment
  - To comment a single line
  - Line start with //
- Multi-line Comment
  - To comment many lines
  - Start with /* and end with */
- Documentation Comment
  - Multi-line comment .
  - A **doc comment** appears immediately before a class, interface, method, or field definition and contains **documentation** for that class, interface, method, or field.
  - To document the API of the code.
  - Start with /* and end with */

Comments

- Single line comment
```
//This is single line comment
```
- Multi-line Comment
```
/* this is multiline comment.
   and support multiple lines*/
```
- Documentation Comment
```
/**
*this is documentation
 *comment
*/
```

Comments



The single line comment is used to comment only one line.

//

The multi line comment is used to comment multiple lines of code.

/*

*/

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

/**   */

Package Statement

- A package is **a group of classes that are defined by a name.**
  - That is, if you want to ***declare many classes within one element***, then you can declare it within a package
- If you do not want to declare any package, then there will be no problem with it, and you will not get any errors.(Package is optional)
- Package is declared as:

package package_name;

Eg: package mypackage;

We can create a package with any name.

Import Statement

- If we want to **use a class of another package**, then we have to import it directly into your program using **import statement.**
- Once imported, a class can be referred to directly, using only its name.
- **Syntax**

import packagename;

Import Statement(contd.)

- Many predefined classes are stored in packages in Java.
- We can import a specific class or classes in an import statement.
- Examples:

import java.io.*; *// import **all** classes from java.io package*

import java.util.Date; //imports the Date class

import java.applet.*; /*imports **all** the classes from the *java.applet* package*/

Interface Statement

- This statement is used to specify interface in Java.
- Interfaces are like a class that includes a group of method declarations.
- It's an optional section and can be used when programmers want to implement multiple inheritances within a program.

Class Definition

- A Java program may contain several class definitions.
- Classes are the **main and essential elements** of any Java program.
- A class is a collection of data and methods.
- It is a good convention to start class name with capital letter
- E.g

class Example{

}

Main Method Class

- Every Java stand-alone program requires the main method as the starting point of the program.
  - This is an essential part of a Java program.

- There may be many classes in a Java program, and **only one class defines the main method**.
- Methods contain data type declaration and executable statements.

```
 class Classname
 {
     public static void main(String args[])
     {


     }
     }
}
```

Main function

```
public static void main(String args[])
{


}
```

- **public** means it **can also be used by code outside of its class.**
- **static** is used when **we want to access a method without creating its object.**
- **void** indicates that a **function does not return a value**.
- **main is the function name**
- **String is a class.**
  - **args is an array of instances of String**

public static void main

- When the main method is declared **public**, it means that it **can also be used by code outside of its class**.
- The word ***static*** is used when ***we want to access a method without creating its object*** (because we call the main method, before creating any class objects*)*
- The word **void** indicates that a **function does not return a value**.
  - main( ) does not return a value.
  - *main() is the starting point of a Java program.*
- Proper spaces can be given in program to make program *easier to read and understand* -It is called *indentation.*

String args[ ]

- **String args[ ]** is an array of instances of class String.
  - Each element is a string, which has been named as "args".
- *We can use any name instead of args.*
- If we Java program is run through the console, we can pass the input parameters (command line arguments), to main( ) method.
  - args **receives any command-line arguments present when the program is executed.**
- **String args[ ]** can also be written as **String[] args**

System.out.println();

- System.out.println();
- This statement is used to **print a text on the screen** as output .
- **System** is a ***predefined class.***
- **out** is an ***object of the PrintWriter class*** defined in the system
- The method **println()** *prints the text that is inside the ( ) on the screen and add a new line.*
- We can also use **print**() method instead of println() method.(new line will not be added)
- All Java statement ends with a semicolon.

Simple Java program

FIRST SIMPLE PROGRAM

```
/*
This is a simple Java program.
 Save this file "Example.java".
*/
import java.io.*;
class Example
{
// Your program begins with a call to main().
      public static void main(String args[])
      {
         System.out.println("This is a simple Java program.");
      }
}
```

- This program begins with the following lines:

```
/*
This is a simple Java program.
 Save this file "Example.java".
*/
```

- *This is a multiline comment.- more than one lines are commented. Lines inside /* and */ are comments*
  - *NOT EXECUTABLE CODES*
- The contents of a comment are **ignored** by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code.

- The next line of code in the program is :

  import java.io.*;

This statement imports all classes in the java.io package.

During compilation , the compiler will look at those classes.

- java.io.* is **Input/Output package**

- The next line of code in the program is :

```
class Example
{
```

- This line uses the keyword **class** to declare that a new class is being defined.
  - *Here Example is the* identifier that is the name of the class.
- The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

- The next line in the program is the ***single-line comment***,

// Your program begins with a call to main().

- Next line

public static void main(String args[])

This is the main function header

- Main function definition

```
public static void main(String args[])
{
    System.out.println("This is a simple Java program.");
}
```

- **System.out.println**("This is a simple Java program.");
  - prints the line **This is a simple Java program.** as output on the screen and control goes to new line

Simple Java program

FIRST SIMPLE PROGRAM

```
/*
This is a simple Java program.
 Save this file "Example.java".
*/
import java.io.*
class Example
{
// Your program begins with a call to main().
      public static void main(String args[])
      {
          System.out.println("This is a simple Java program.");
      }
}
```

```
D:/OOP> javac Example.java
D:/OOP> java Example
This is a simple Java program.
```

Garbage Collection in Java

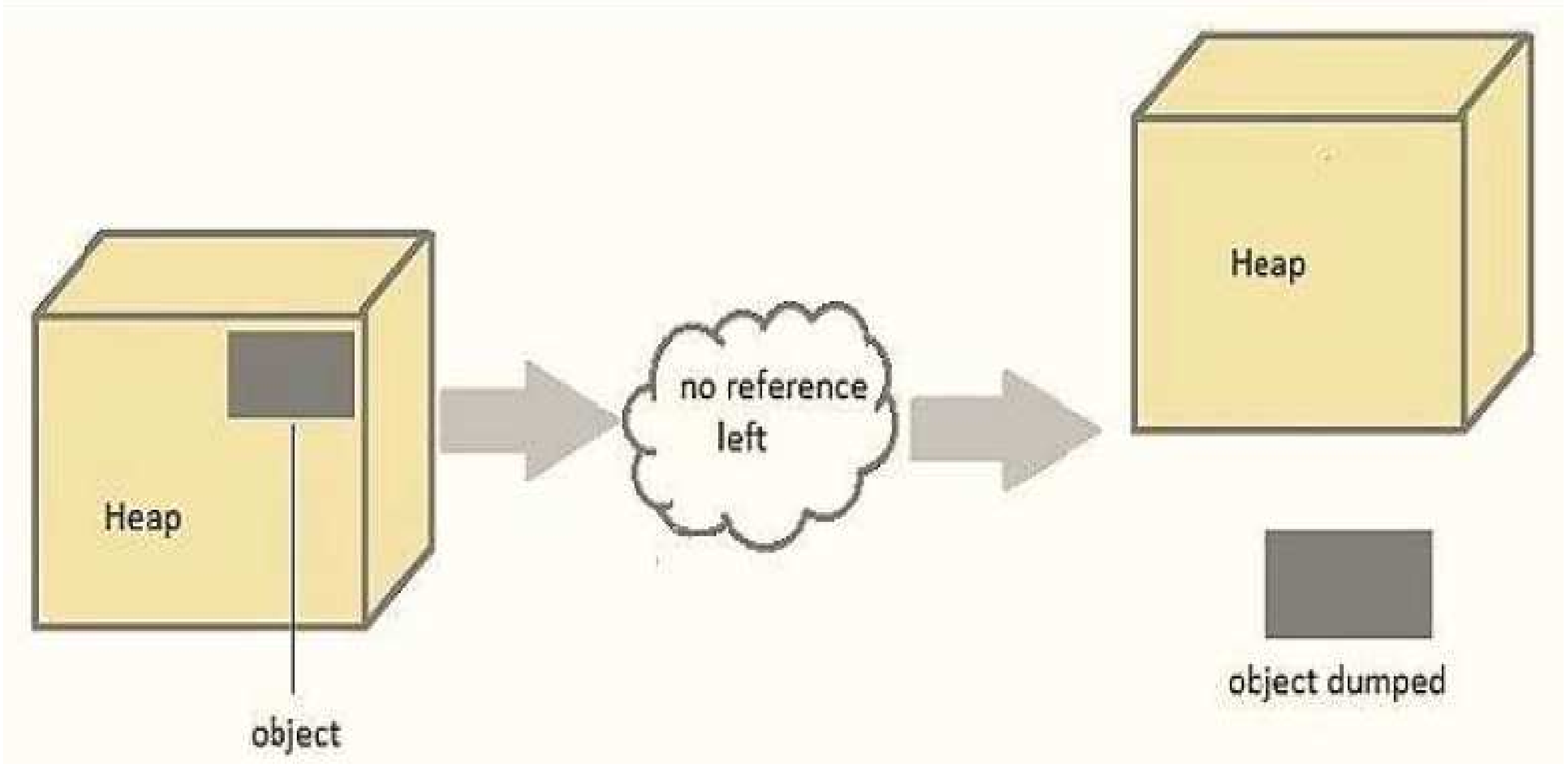
- Garbage collection is a process of **releasing unused memory.**
  - objects can be **dynamically allocated using the new operator** which can be **automatically deallocated using garbage collector.**
- When **no references to an object** exist, that **object** is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
  – The technique that accomplishes this is called *garbage collection.*

Garbage Collection in Java

- When JVM starts up, it creates a **heap area** which is known as runtime data area. This is where all the objects (instances of class) are stored.
- Since this heap area is limited, it is required to ***manage this area efficiently*** by **removing the objects that are no longer in use**.

Garbage collection(contd.)

- The process of **removing unused objects from heap memory** is known as **Garbage collection**
  - this is a part of ***memory management*** in Java.
- Languages like C/C++ don’t support automatic garbage collection,
- In Java, the **garbage collection is automatic.**


Heap
object
no reference left
Heap
object dumped

Garbage collection(contd.)

- In java, **garbage** means ***unreferenced objects***.
- Main objective of Garbage Collector is to <u>free heap memory by destroying unreachable objects.</u>
- **Unreachable objects :** An object is said to be unreachable if and only if(iff) it **doesn’t contain any reference to it**.
- Eligibility for garbage collection : An object is said to be eligible for GC(garbage collection) iff it is unreachable.

Request for Garbage Collection

- We can request to JVM for garbage collection however, it is upto the JVM when to start the garbage collector.
- Java **gc() method** is used to **call garbage collector explicitly**.
  - However gc() method does not guarantee that JVM will perform the garbage collection.
  - It only **request the JVM for garbage collection**. This method is present in System and Runtime class.

finalize() method

- **finalize()** method – This method is **called** each time just **before the object is garbage collected** and it **perform cleanup processing.**
  - By using finalization, we can **define specific actions** that will occur when an object is just about to be reclaimed by the garbage collector.
  - E.g if an object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed.

Garbage collection(contd.)

- The Garbage collector of JVM collects **only those objects that are created by new keyword.**
- So if we have created any object **without new**, we can use **finalize() method** to **perform cleanup processing**

Lexical Issues

- Java programs are a collection of JAVA **TOKENS**
  - Whitespace
  - Identifiers
  - Literals
  - Comments
  - Operators
  - Separators
  - Keywords.

Lexical Issues- Whitespace

- Java is a **free-form language**.
  - We ***do not need to follow any special indentation rules***.
  - It means that we can write statements(code) in java program **all on one line** or **in any other strange way.**
- There should be **at least one whitespace character between each token** (if it is not already delineated by an operator or separator).
- In Java, **whitespace** is a
  - **Space**
  - **Tab**
  - **Newline.**

Lexical Issues-Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of
  - **Uppercase letters ABCD……Z**
  - **Lowercase letters abcd…z**
  - **Numbers 0123456789**
  - **Underscore _**
  - **Dollar-sign characters $**
- They <u>must not begin with a number.</u>
- Java is case-sensitive

Lexical Issues-Identifiers

AvgTemp	Valid
count	Valid
2count	Invalid. should not begin with number
a4	Valid
Not/ok	Invalid. / not allowed
$test	Valid
this_is_ok	Valid
high-temp	Invalid - not allowed

Lexical Issues- Comments

- Single line
- Multi-line
- Documentation comment

Lexical Issues- Literals

- A constant value in Java is created by using a *literal representation of it*.
- *Example of* literals:

100	integer
98.6	floating point
'X'	character
“This is a test”	string

- A literal can be used anywhere a value of its typeis allowed.

Operators

- Operators are used for performing operations.
  - **Arithmetic Operators**
    - **+, -, *, /, %**
    - **++, +=, -=,*=, /=, %=, - -**
  - **Bitwise Operators** ~ Bitwise unary NOT
    - &,| ,^
    - >> ,>>> ,<<
    - &=, |=, ^=, >>=, >>>=
  - **Relational Operators**
    - = =, !=, >, < , <=, >=
  - **Logical Operators**
    - **&, |, ^, ||,&&,** &=,| =, ^=, ==, !=, ?:

Lexical Issues- Separators

- In Java, there are a few characters that are used as separators.
- Separator is a symbols that is used to **separate a group of code from one another.**
- The most commonly used separator in Java is the **semicolon**. **;**

Lexical Issues- Separators(contd.)

The separators are

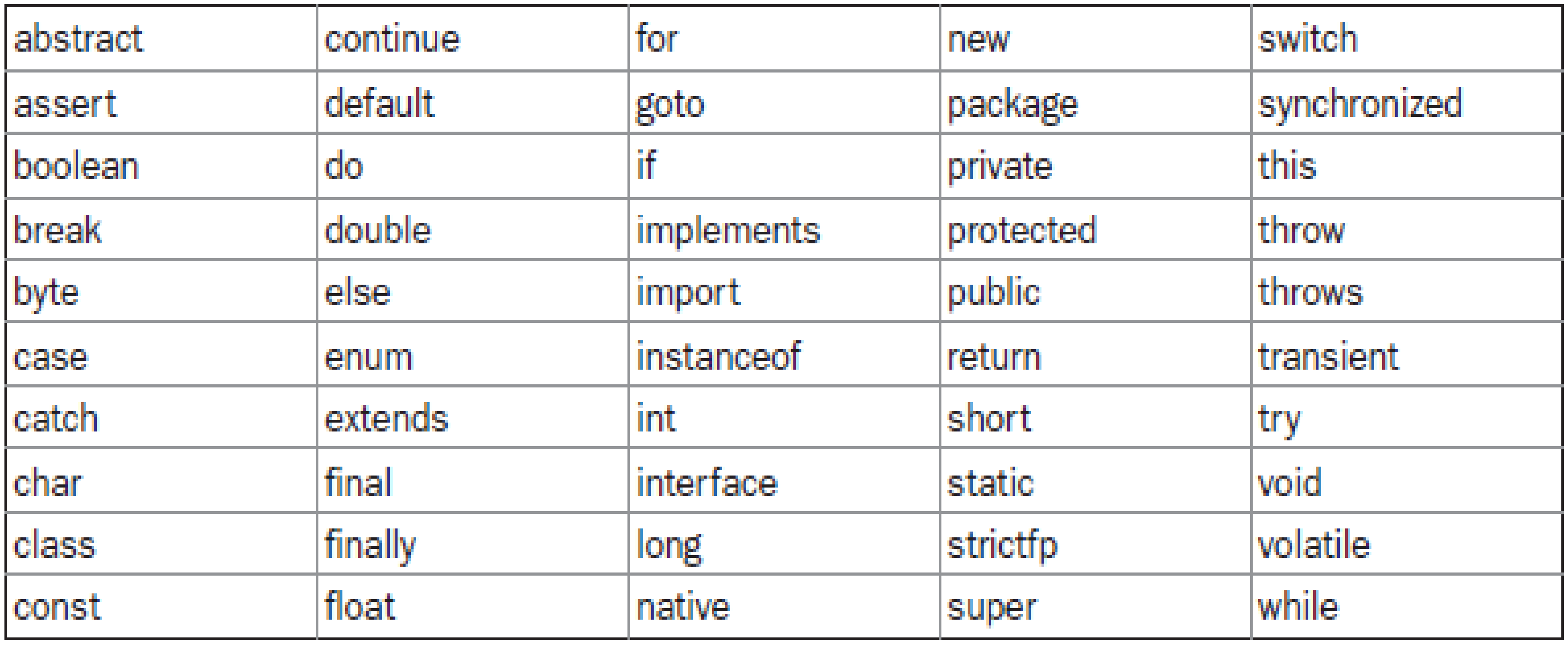
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Lexical Issues- The Java Keywords

- Keywords are **reserved words.**
- There are 50 keywords currently defined in the Java language.
- These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.
- These **keywords cannot be used as names for a variable, class, or method.**

Lexical Issues- The Java Keywords

- **Java keywords**



abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Special values

- In addition to the keywords, Java reserves the following:
  - **true, false, null.**
- These are **values** defined by Java.
- We should not use these words for the names of variables, classes, and so on.

REFERENCE

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

CS205 Object Oriented Programming in Java

Module 1 - **Introduction**

(UML diagrams, Use case model)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Core Java Fundamentals:
  - ✓UML diagrams
    - ✓ Use case model

UML

- UML (Unified Modeling Language) is a general-purpose, **graphical modeling language** in the field of Software Engineering.
- Developed to help system and software developers for specifying, visualizing, constructing and documenting the artifacts of software systems
- UML was created by the Object Management Group (OMG)
- UML is not a programming language.
- UML is a **visual language for developing software blueprints** (designs).
  - For example, while constructing buildings, a designer or architect develops the building blueprints. Similarly, we can also develop blueprints for a software system.

Advantages of using UML

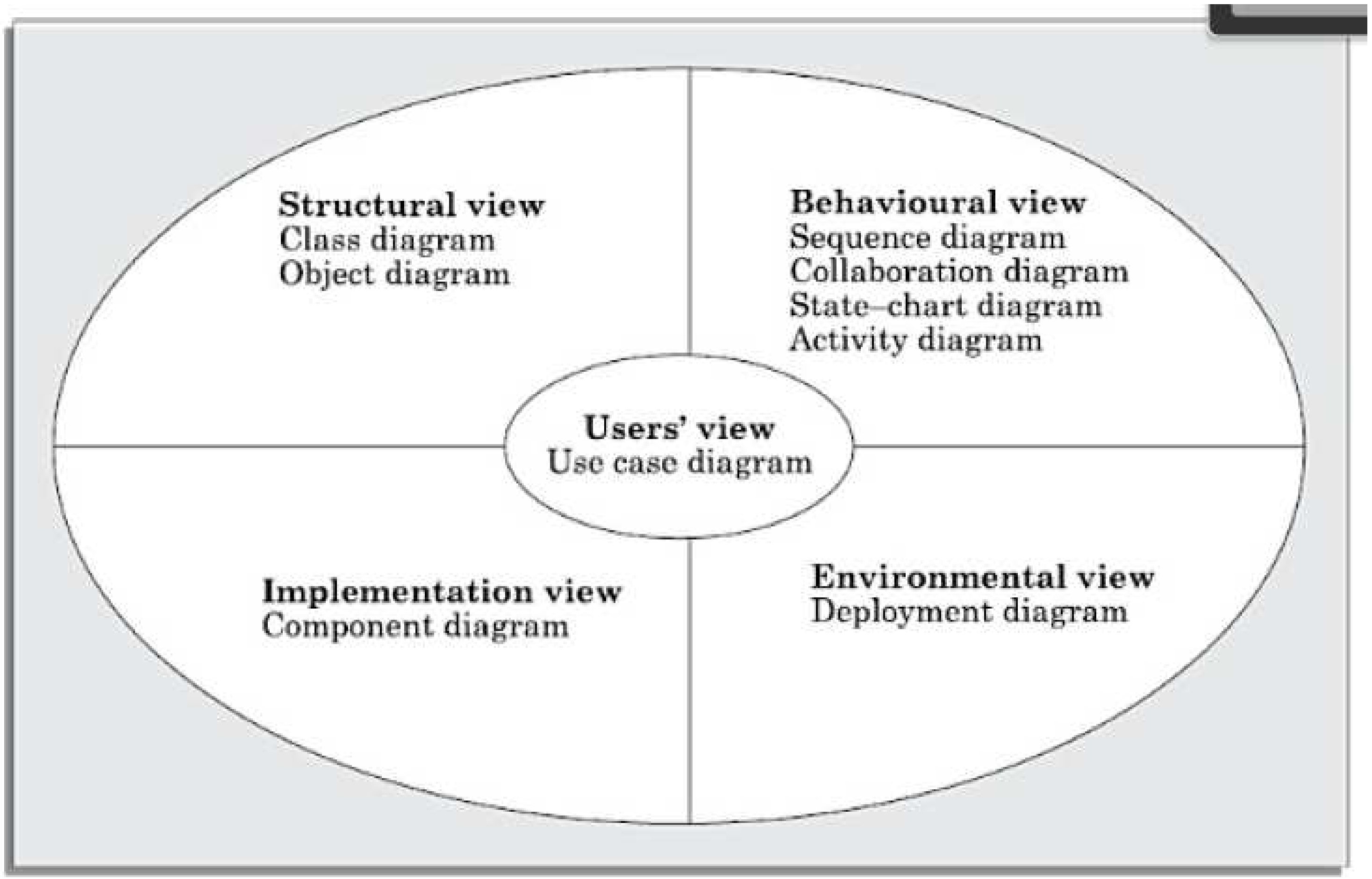
- Advantages of using UML
- □ Readability and Reusability
- □ Provides standard tool for software design
- □ Helps reduce development cost
- □ Makes communication and sharing ideas among programmers quite easy
- □ Unification of all modelling concepts

UML DIAGRAMS

- A **UML diagram** is a **diagram** based on the **UML** (Unified Modeling Language)
- UML diagrams provide **different perspectives** of a software system to be developed.
- Each perspective focuses on some specific aspect and ignores the rest.
- UML diagrams helps to have a **comprehensive understanding of the system.**

UML diagrams

- UML diagrams can capture the following views (models) of a system:
  - User’s view
  - Structural view
  - Behaviourial view
  - Implementation view
  - Environmental view



- Different types of diagrams and views supported in UML.

Views of a system

- The **users' view** captures the view of the system in terms of the <u>functionalities offered by the system to its users.</u>
- The **structural view** defines the structure of the problem (or solution) in terms of the kinds of objects (classes). It also captures the relationships among the classes (objects).

Views of a system(contd.)

- The **behaviourial view** captures **how objects interact with each other** in time to realize the system behaviour. It captures the **time-dependent (dynamic) behaviour** of the system.
- This **implementation view** captures the important components of the system and their interdependencies.
- **Environmental view** models how the different components are implemented on different pieces of hardware

UML diagram

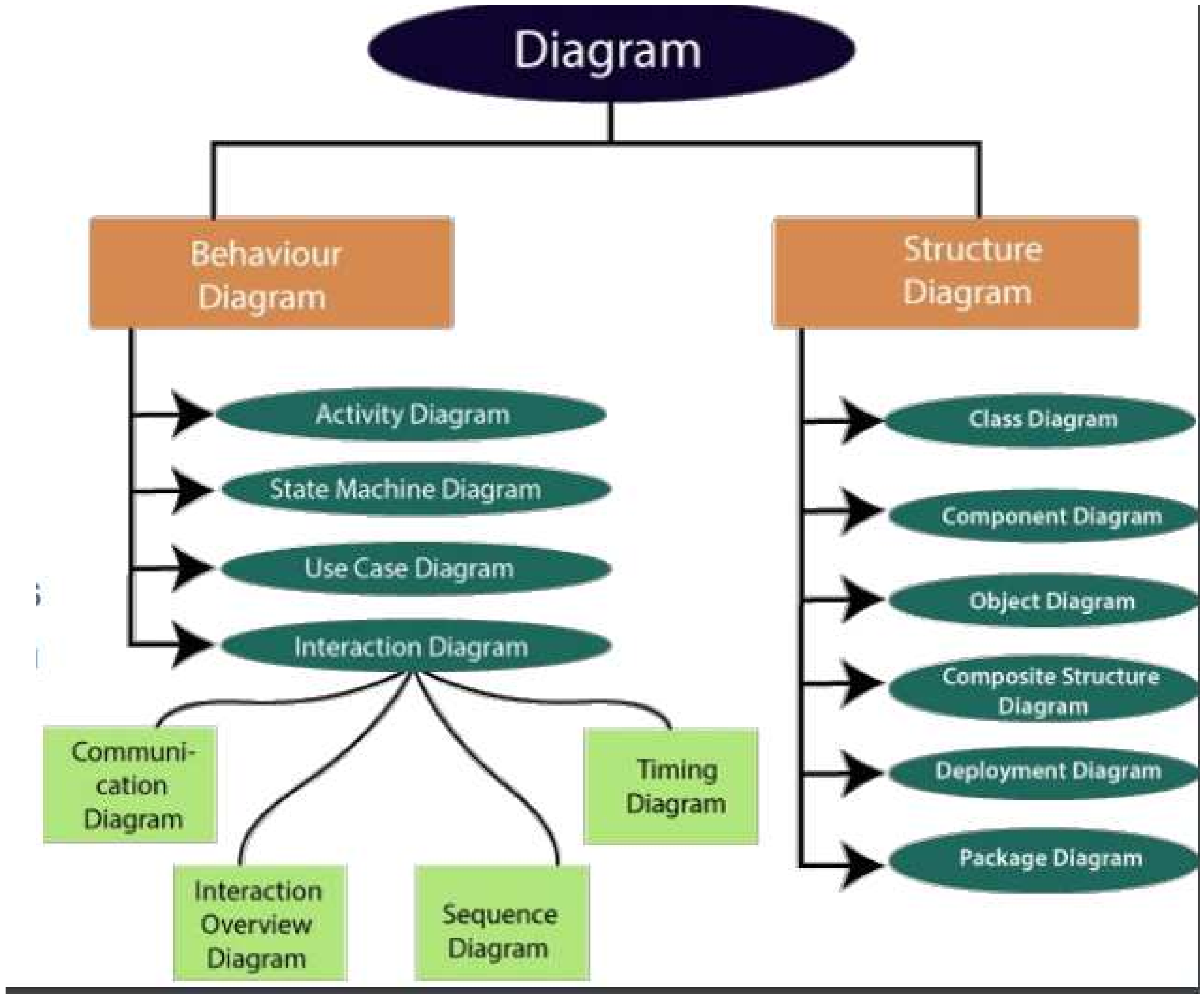
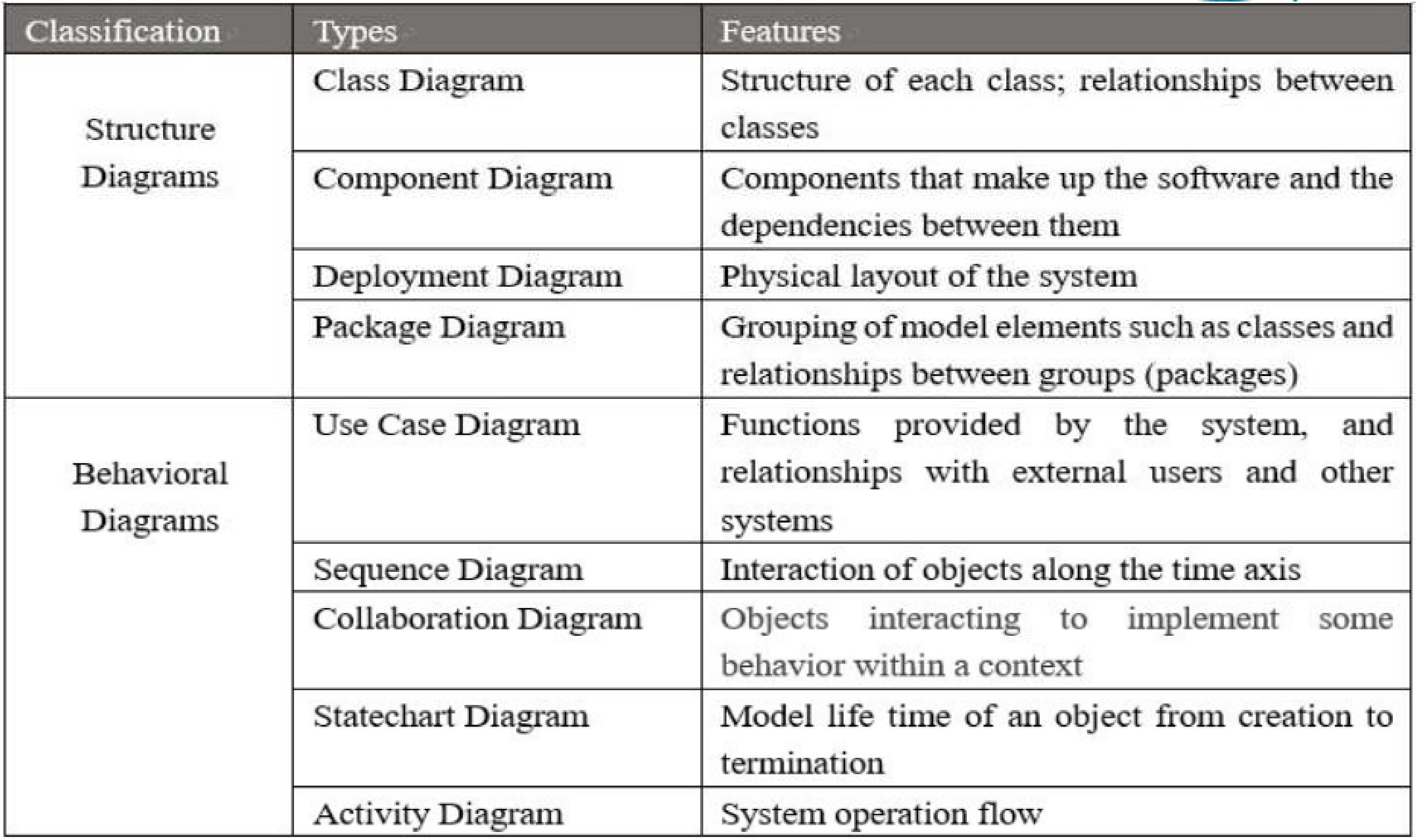


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Use Case Diagram
Interaction Diagram
Communi-cation Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram
Class Diagram
Component Diagram
Object Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram



Classification	Types	Features
Structure Diagrams	Class Diagram	Structure of each class; relationships between classes
	Component Diagram	Components that make up the software and the dependencies between them
	Deployment Diagram	Physical layout of the system
	Package Diagram	Grouping of model elements such as classes and relationships between groups (packages)
Behavioral Diagrams	Use Case Diagram	Functions provided by the system, and relationships with external users and other systems
	Sequence Diagram	Interaction of objects along the time axis
	Collaboration Diagram	Objects interacting to implement some behavior within a context
	Statechart Diagram	Model life time of an object from creation to termination
	Activity Diagram	System operation flow

Use Case Diagram

- Use case diagrams represents the **functions** provided by the system and the **relationships with external users or system.**

Use Case Diagram

- The use case model for any system consists of a **set of use cases.**
- **Use cases** specify the **expected behavior or functions** (what it will do), and not the exact method of making it happen (how).
- The purpose of a use case is to define a piece of coherent **behaviour** without revealing the internal structure of the system.
- The use cases do not mention any specific algorithm to be used nor the internal data representation or internal structure of the software.

Use Case Diagram (contd.)

- Use case specifies **what are the actions that can be done by users** in that system.

- Library information system (LIS), the use cases could be:
  - issue-book
  - query-book
  - return-book
  - create-member
  - add-book, etc.

Use Case Diagram (contd.)

- A use case consists of one **main line sequence** and **several alternate sequences.**
  - The **main line sequence** represents the ***interactions*** between a user and the system ***that normally take place***.
  - Several ***variations*** to the main line sequence (called ***alternate sequences***) may also exist.
- The main line sequence and each of the variations are called **scenarios or instances** of the use case.

Use Case Diagram (contd.)

- E.g Withdraw cash form ATM
  - In the **mainline sequence,** use case supported by a ATM would be
    - the user inserts the ATM card, enters password, selects the amount withdraw option, enters the amount to be withdrawn, completes the transaction, and collects the amount.
  - The **variations or alternate sequences** can be:
    - Password is invalid.
    - The amount to be withdrawn exceeds the account balance.

Use Case Diagram (contd.)

- Use case diagrams only summarizes **some of the relationships between use cases, actors, and systems.**
  - It does not show the order in which steps are performed to achieve the goals of each use case.
- Use cases correspond to the high-level functional requirements.

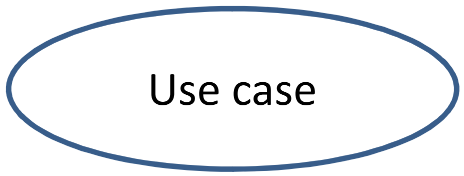
Use Case Diagram (contd.)

- Use case model represents a **functional or process model** of a system.
- Each use case is independent of the other use cases.
- But implicit ***dependencies*** may exist among use cases because there may be dependencies among use cases at the implementation level due to factors such as shared resources, objects, or functions.
  - For example, In the Bookshop Automation Software, ***update_inventory*** and *sale_book* are two independent use cases. But, during execution of *sale_book* there is an implicit dependency on u*pdate_inventory*. If sufficient quantity of book is unavailable in the inventory, *sale_book* cannot operate until the inventory is replenished using *update_inventory.(get more books and update the inventory)*

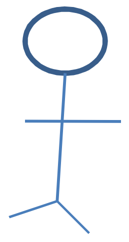
Representation of Use Case diagram

Use Case

- Use case in a system means ***operations*** in a system
- In the use case diagram, each **use case** is represented by an ellipse with the name of the use case written inside the ellipse.



Representation of Use Case diagram(contd.)



Actors

- The different users of the system are represented by using stick person icons. They are called **actors.**
- Both human users and external systems can be represented by stick person icons.
- When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

Representation of Use Cases(contd.)

- An **actor** is a role played by a user with respect to the system use.
  - E.g. Librarian can be actor in Library Management system
- The same user may play the role of multiple actors.
- An actor can participate in one or more use cases.

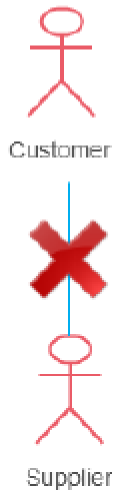
Representation of Use Case diagram(contd.)

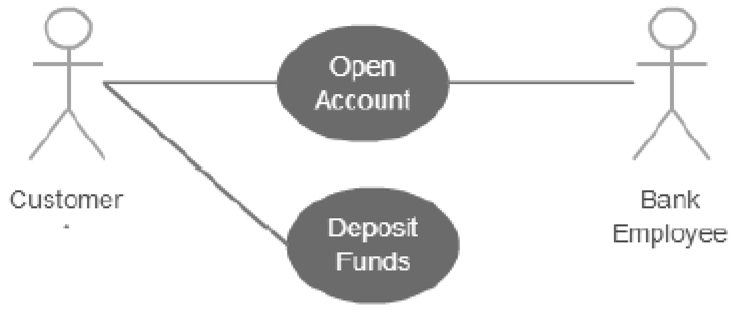
Communication relationship

- The **line connecting an actor and the use case** is called the **communication relationship**.
  - It indicates that an actor makes use of the functionality provided by the use case

Association between actor and use case

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.

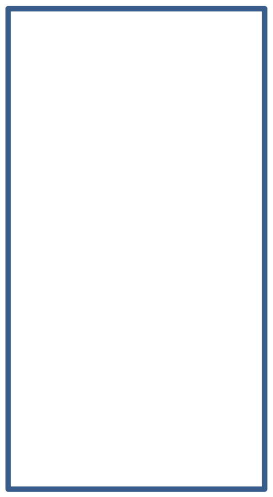




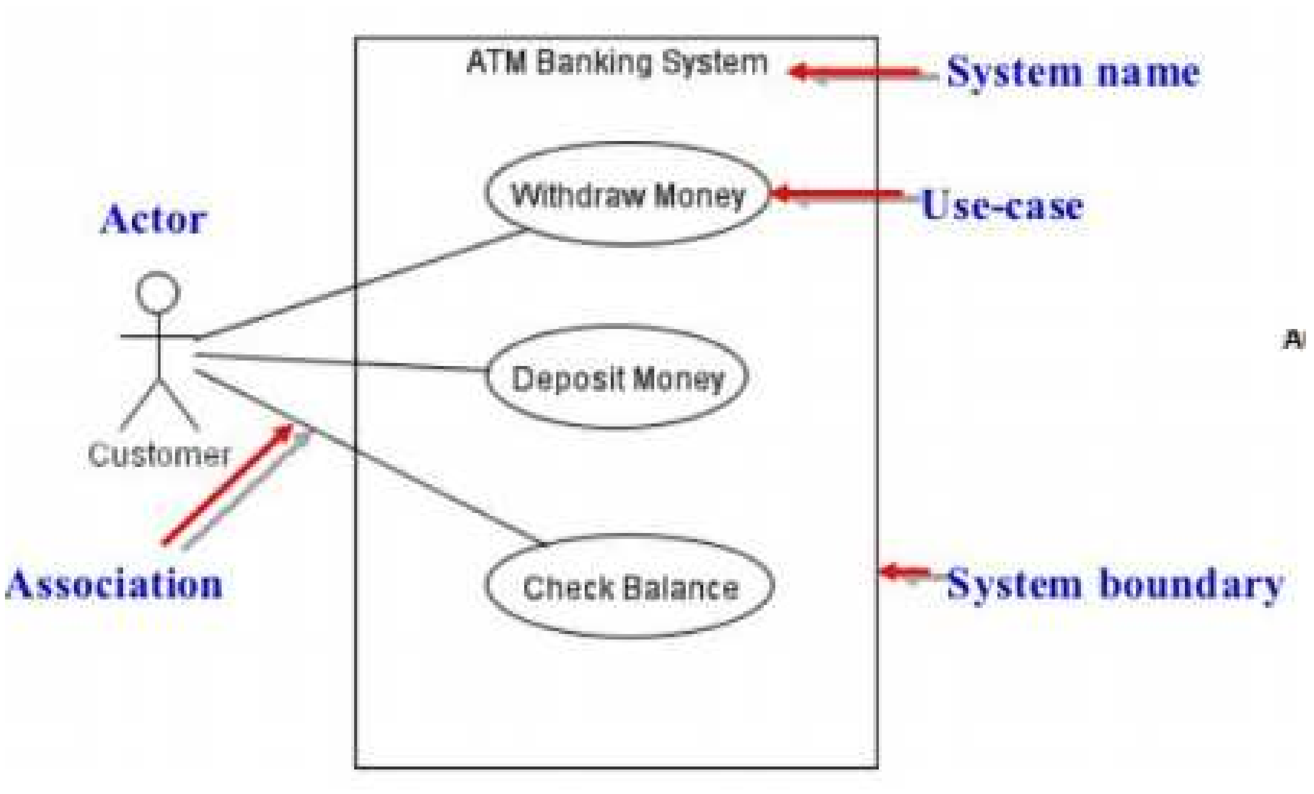
Representation of Use Case diagram(contd.)

System Boundary

- All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary.
- The name of the system being modeled (e.g., library information system) appears inside the rectangle.
- Anything within the box represents functionality that is in scope and anything outside the box is not.
  - However, drawing the system boundary is optional.

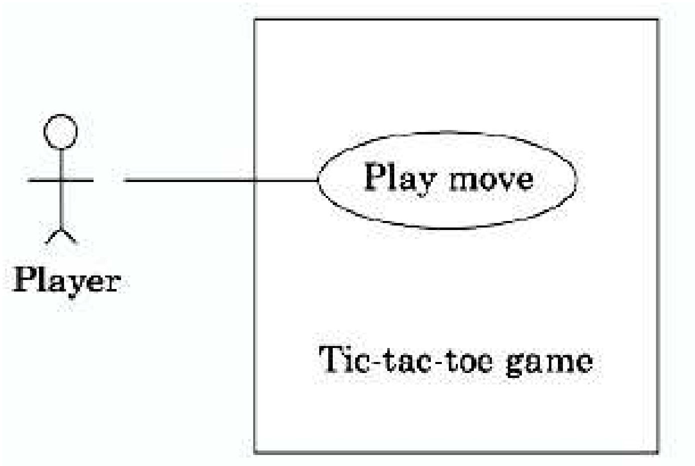


E.g. Use case diagram



Example-Use case diagram

- **Tic-tac-toe game software**



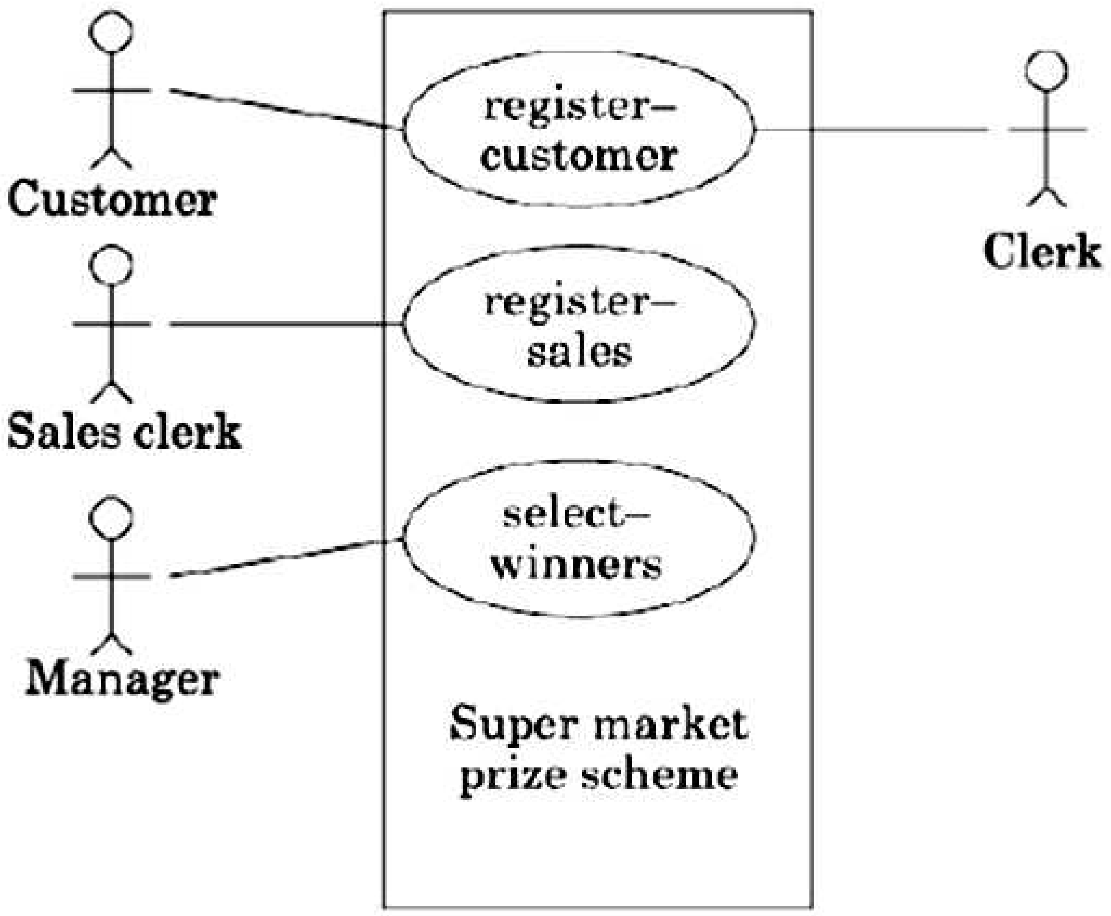
- This software has only one use case, namely, “play move”

E.g. Super market prize scheme

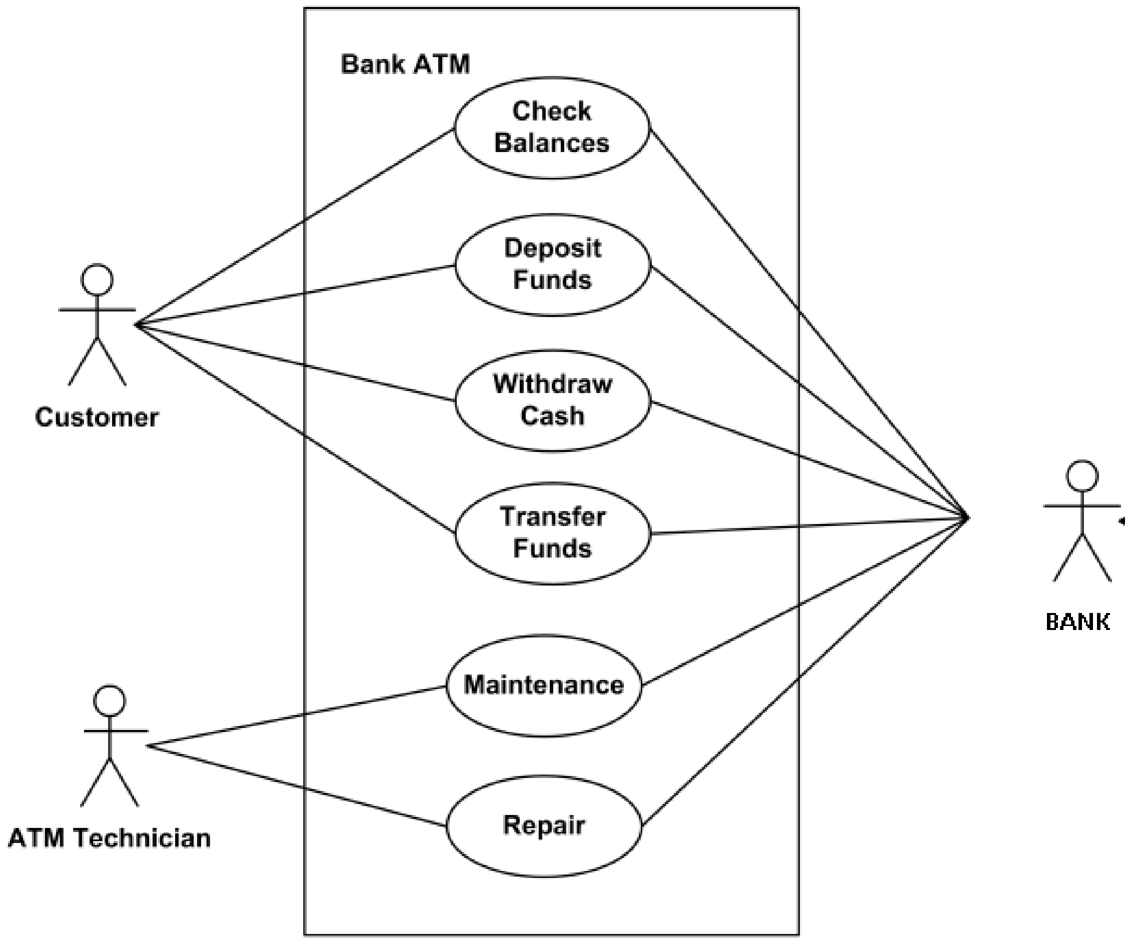
- A super market needs to develop a software that would help it to automate a scheme that it plans to introduce to encourage regular customers. In this scheme, a **customer would have first register** by supplying his/her residence address, telephone number, and the driving license number. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year.

Example-Use case diagram

- Super market prize scheme

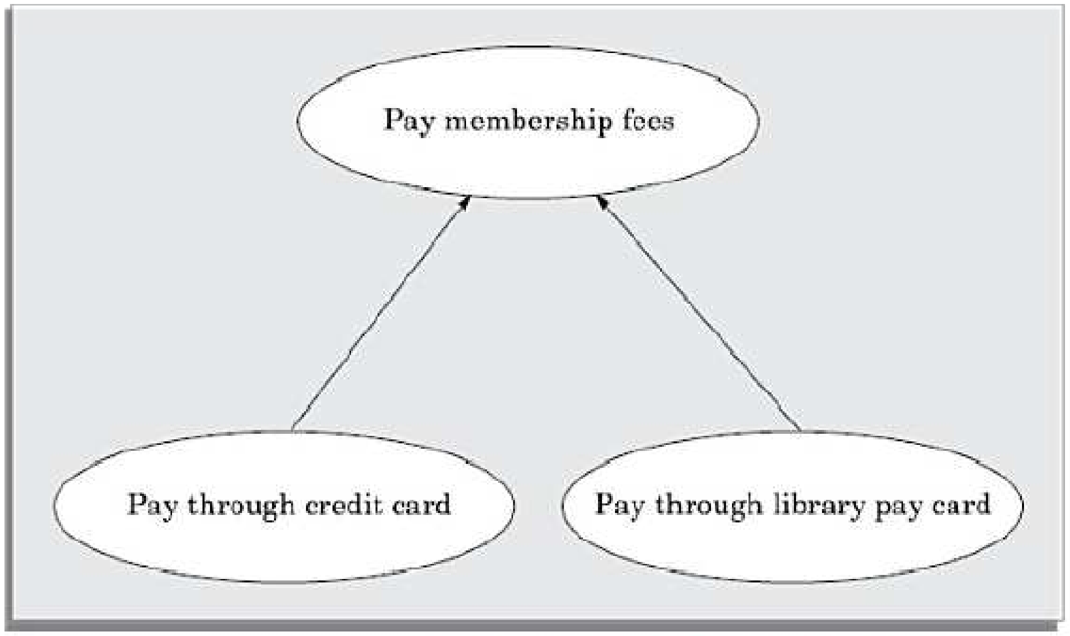


Example-Use case diagram



Generalisation

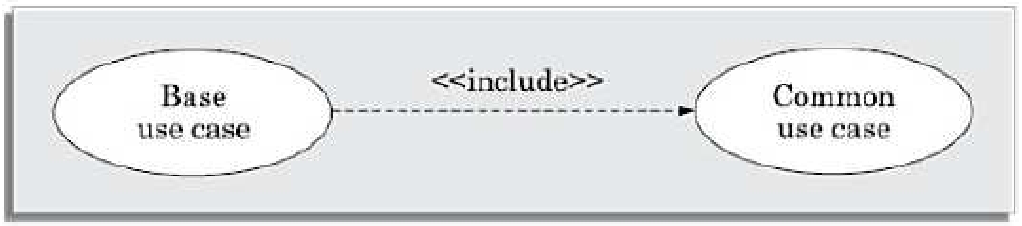
- Generalisation works the same way with use cases as it does with classes.
- The child use case inherits the behaviour and meaning of the parent use case.
- Base and the derived use cases are separate use cases and should have separate text descriptions.

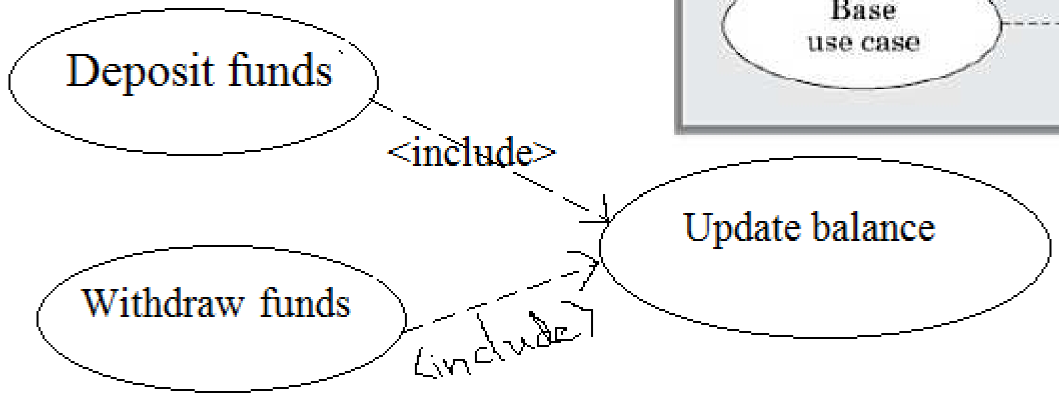


Representation of use case generalisation.

Includes

- The includes relationship in the older versions of UML (was known as the **uses relationship.**
- The includes relationship implies **one use case includes the behaviour of another use case** in its sequence of events and actions.
- The includes relationship is represented using a predefined stereotype <<include>>.
- Include use case is **required** not optional.

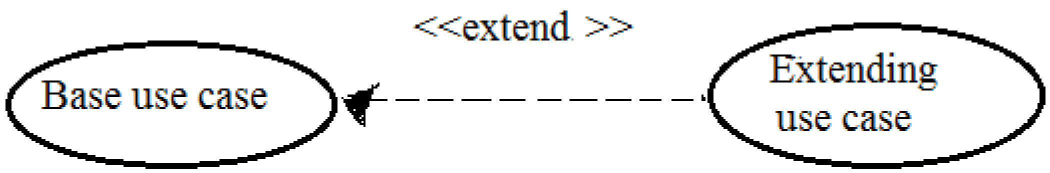




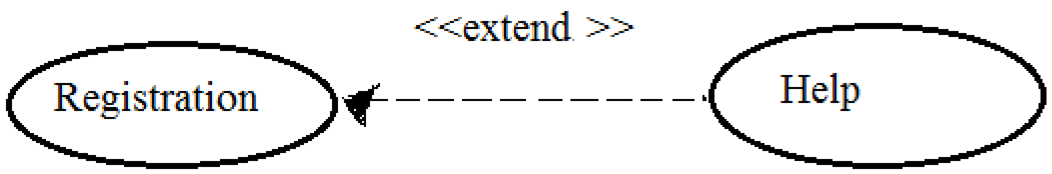
Prepared by Renetha J.B.

Extends

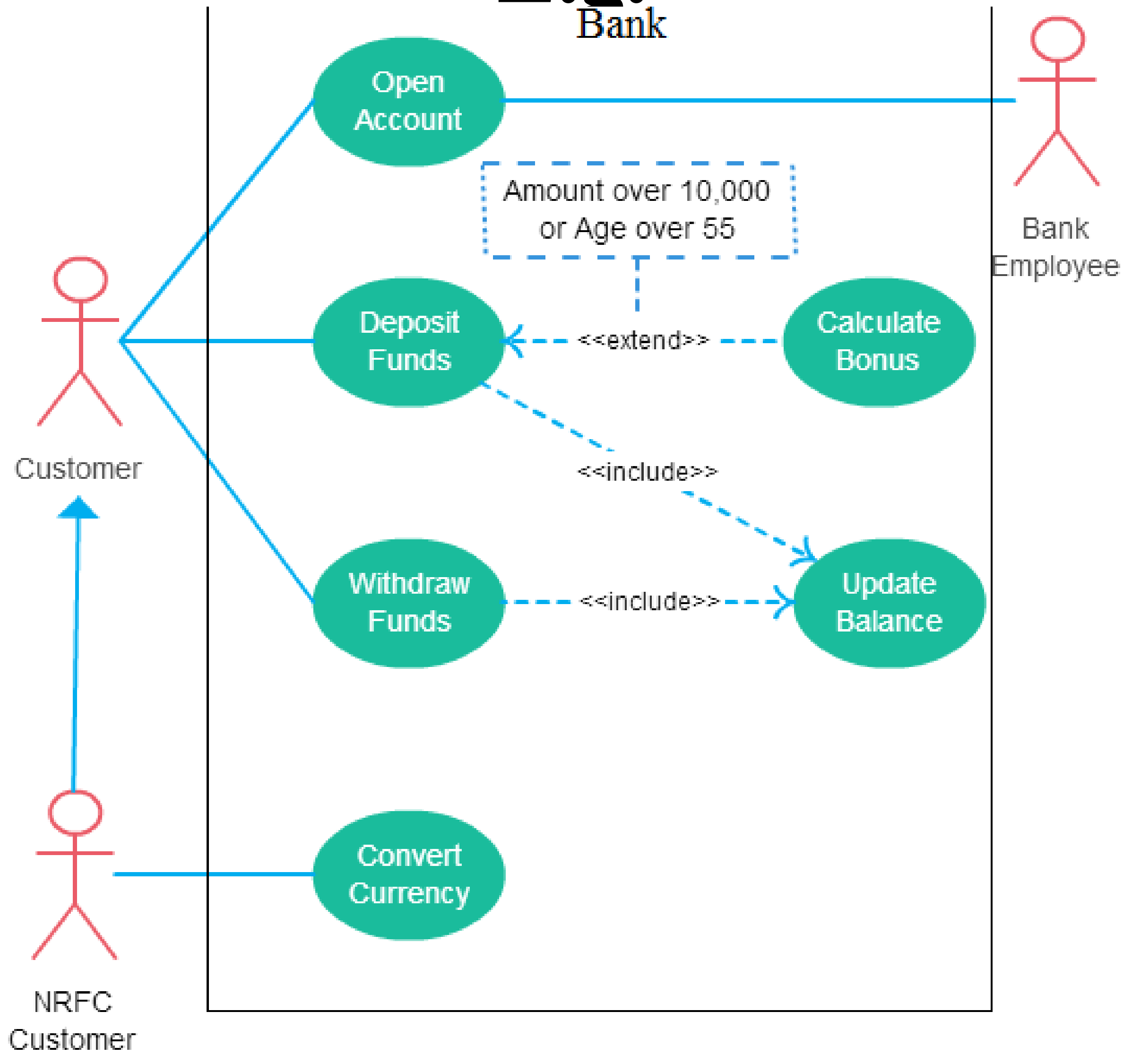
- The main idea behind the extends relationship among use cases is that it allows to show **optional system behaviour**.
  - An ***optional system behaviour is executed*** only if certain conditions hold, otherwise the optional behaviour is not executed.
  - The extends relationship is normally used to capture alternate paths or scenarios.



  - E.g. Help is the extending use case of Registration. Suppose if help option is clicked during Registration then only operation associated with Help is provided

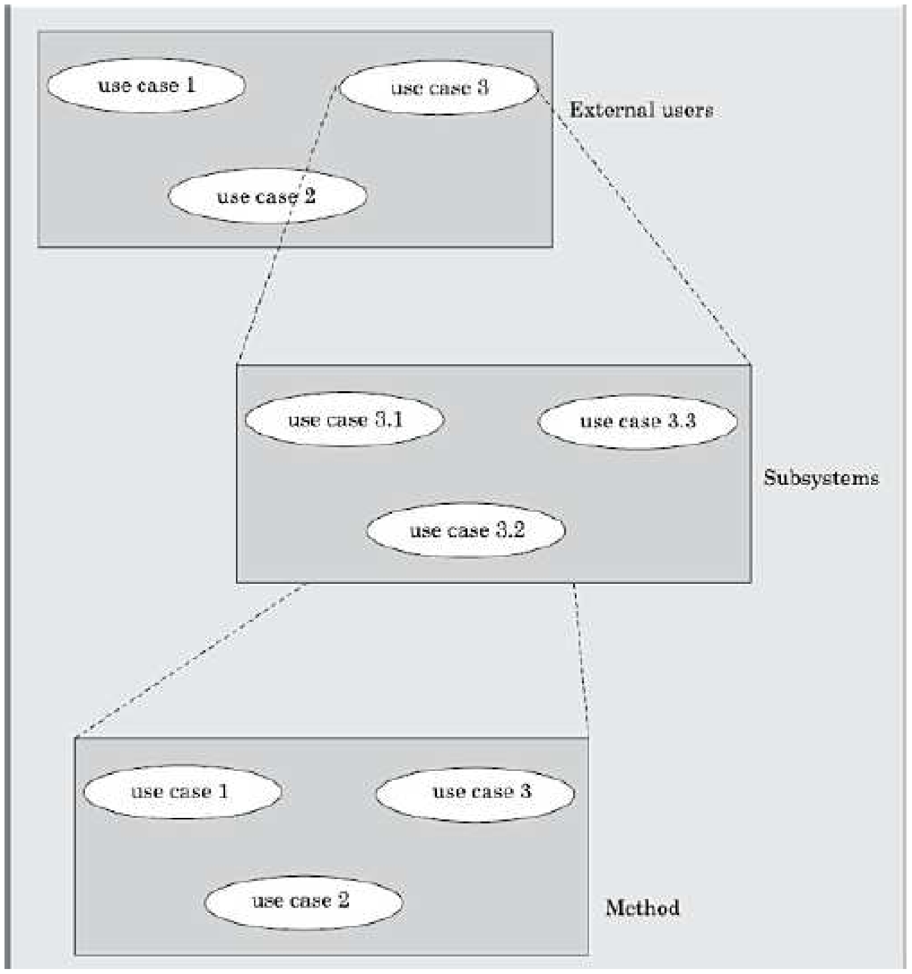


E.g.

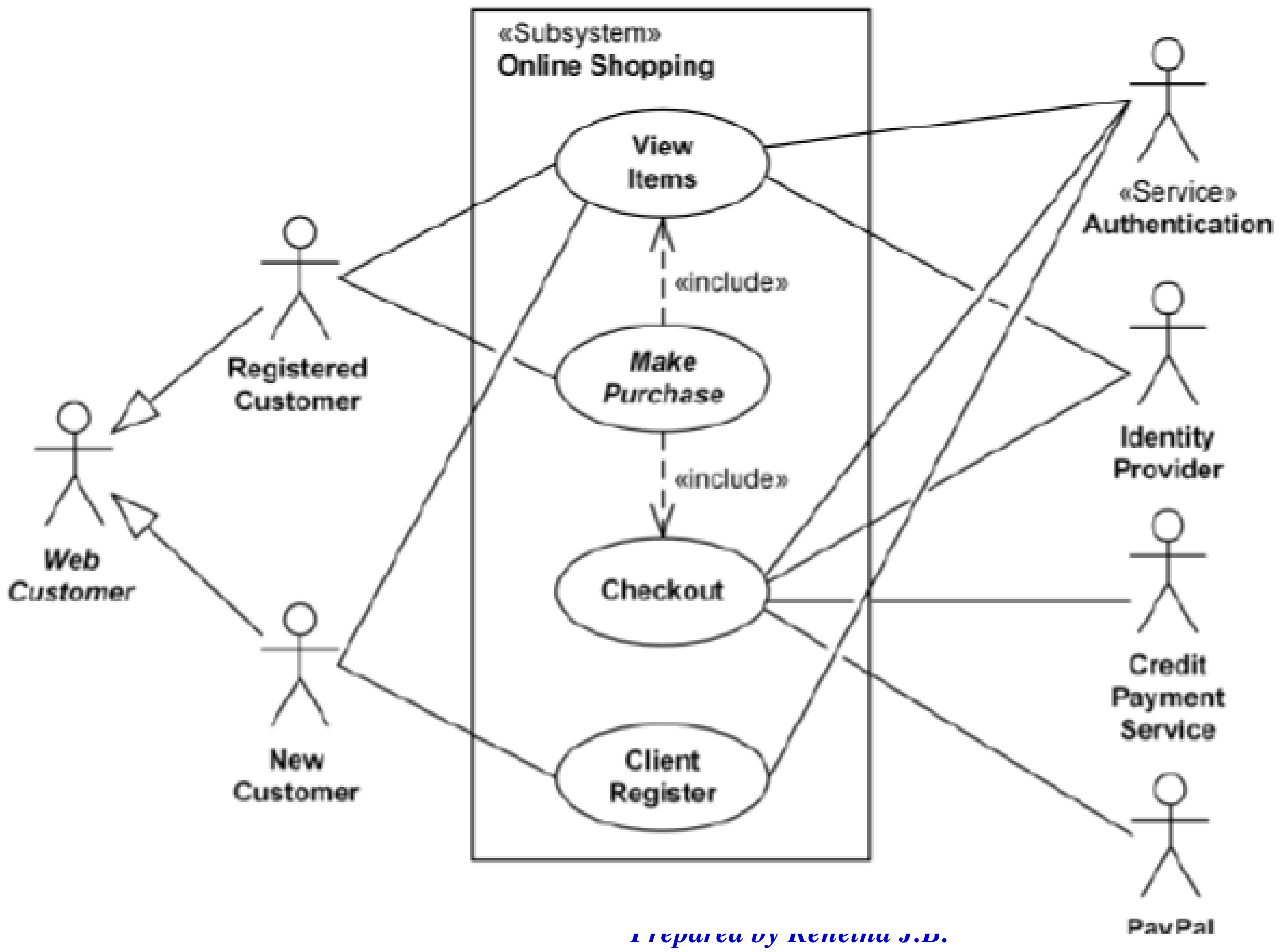

Bank
Open Account
Amount over 10,000 or Age over 55
Bank Employee
Customer
Deposit Funds
<<extend>>
Calculate Bonus
<<include>>
Withdraw Funds
<<include>>
Update Balance
NRFC Customer
Convert Currency

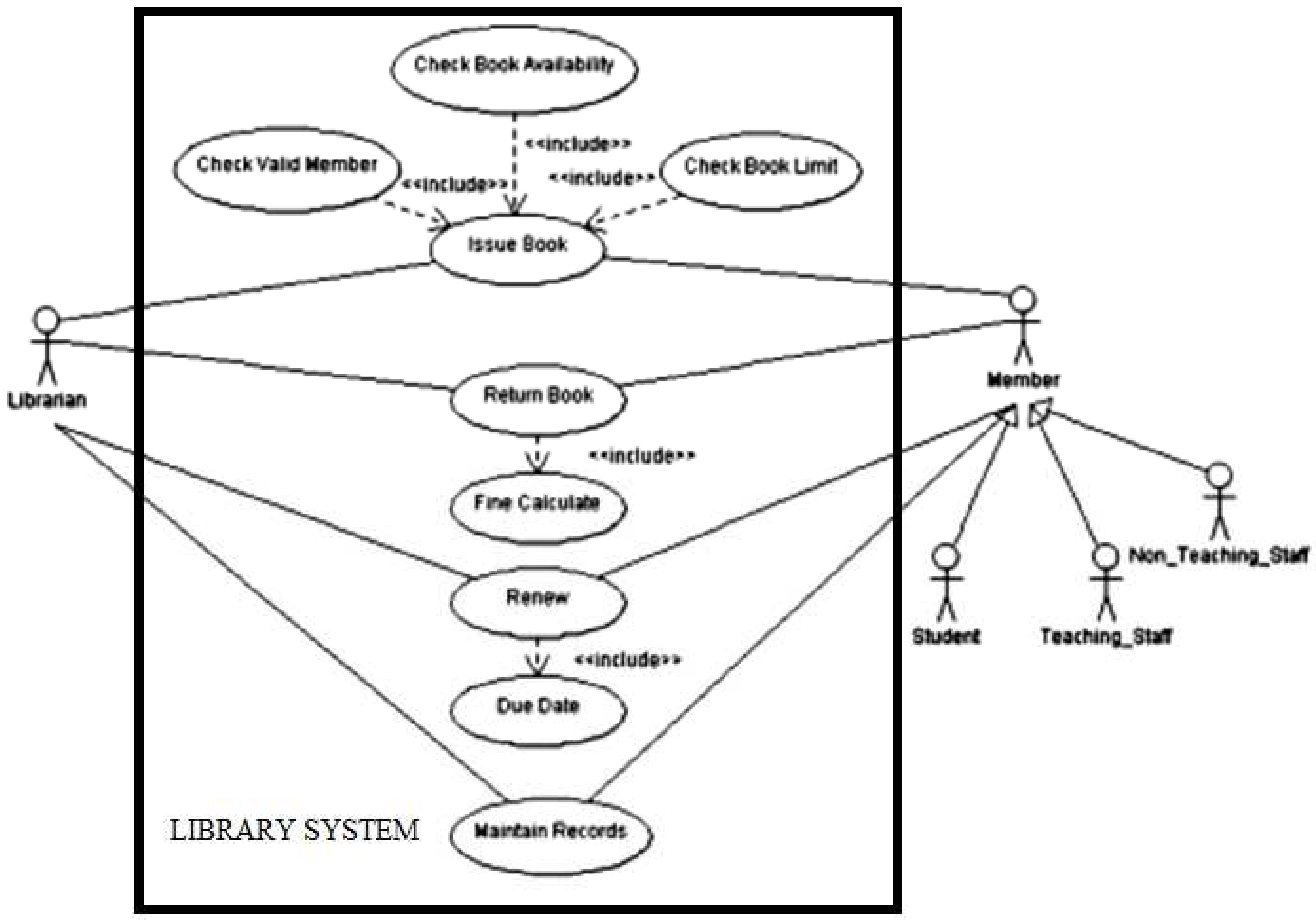
Organisation

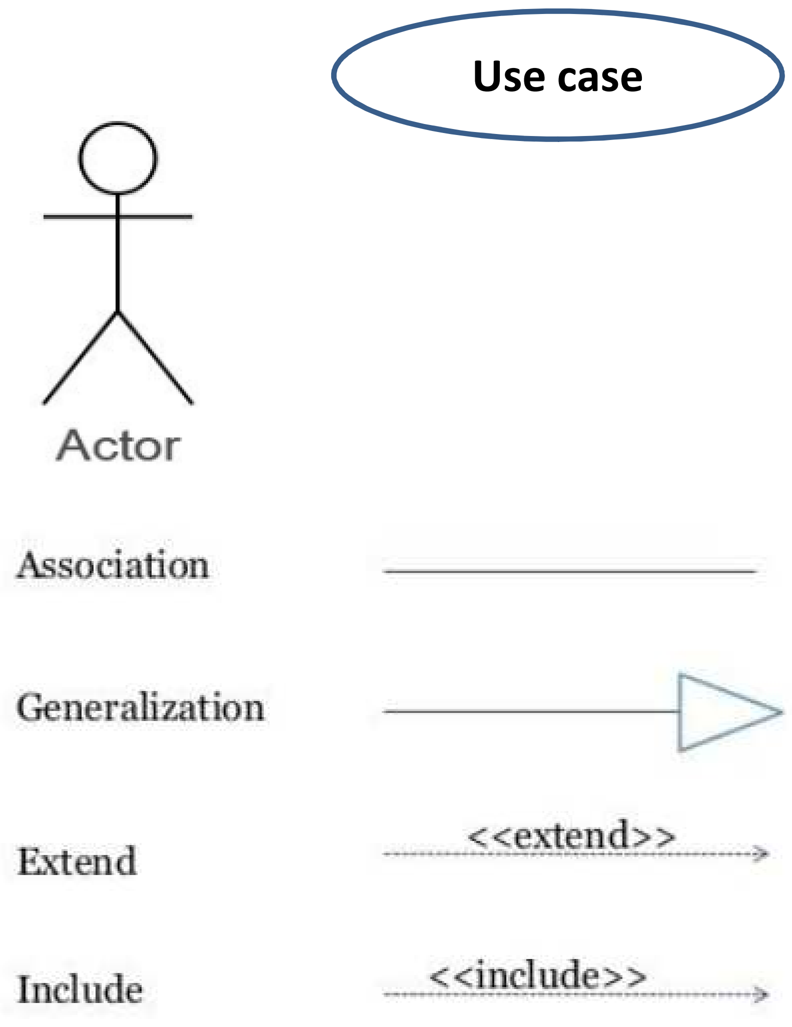
- When the use cases are factored, they are organised hierarchically. The highlevel use cases are refined into a set of smaller and more refined use cases



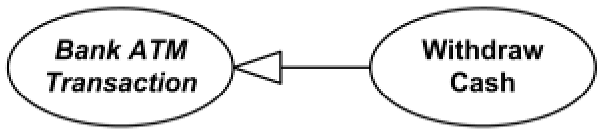
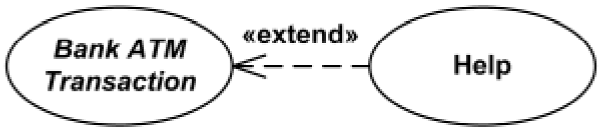
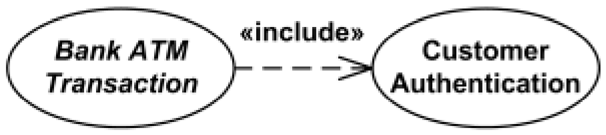
Online shopping-use case diagram




Check Book Availability
Check Valid Member
Check Book Limit
<<include>>
<<include>>
<<include>>
Issue Book
Librarian
Member
Return Book
<<include>>
Fine Calculate
Renew
<<include>>
Due Date
LIBRARY SYSTEM
Maintain Records
Student
Teaching_Staff
Non_Teaching_Staff


Use case
Actor
Association
Generalization
Extend
<<extend>>
Include
<<include>>

Geeralization vs Extend vs Include (Example)

Generalization	Extend	Include
		

Reference

- Rajib Mall, **Fundamentals of Software Engineering, 4th edition, PHI, 2014**

CS205 Object Oriented Programming in Java

Module 1 - **Introduction (**Class diagram, Interaction diagram**)**

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- Introduction:
  - ✓UML diagrams
    - ✓ Class diagram
    - ✓Interaction diagram

Class diagram

- A class diagram describes the **static structure** of a system.
- The static structure of a system comprises a number of class diagrams and their dependencies.
- The main constituents of a class diagram are
  - classes and
  - their relationships — generalisation, aggregation, association, and various kinds of dependencies

Class diagram- CLASSES

- The classes represent group of entities with similar features, i.e., **attributes** and **Operations.**
- A class notation consists of three parts(compartments)
  - First compartment**(mandatory) - Class Name**
  - Second compartment(optional)- **Attributes**
  - Third compartment(optional)-**Operations-**

Class diagram- CLASSES

- First compartment(**mandatory**) **Class Name** -The name of the class is written here.
- Class name has following properties:-
  - is **bold** and ***centered***.
  - use mixed case convention
  - starts with uppercase letter
- Object names are written using a mixed case convention, but starts with a small case letter

Class diagram- ATTRIBUTES

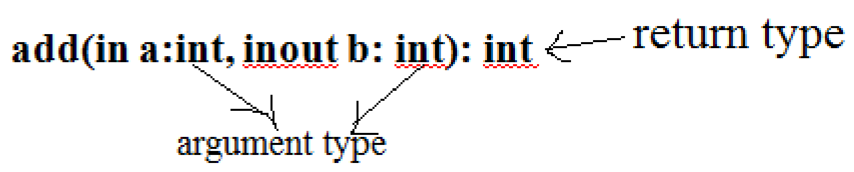
- Second compartment(***optional***)- **Attributes (**property of a class)
  - Attribute names are written **left-justified** using plain type letters, and the names should begin with a **lower case letter**
  - The attribute type can be shown after the colon.
  - The attribute name may be followed by an initialization expression. (initial value)
    - E.g. $\underbrace{\text{rollno}}_{\text{attribute}}:\underbrace{\text{int}}_{\text{type}}=\underbrace{1}_{\text{Initial value}}$

Class diagram- ATTRIBUTES(contd.)

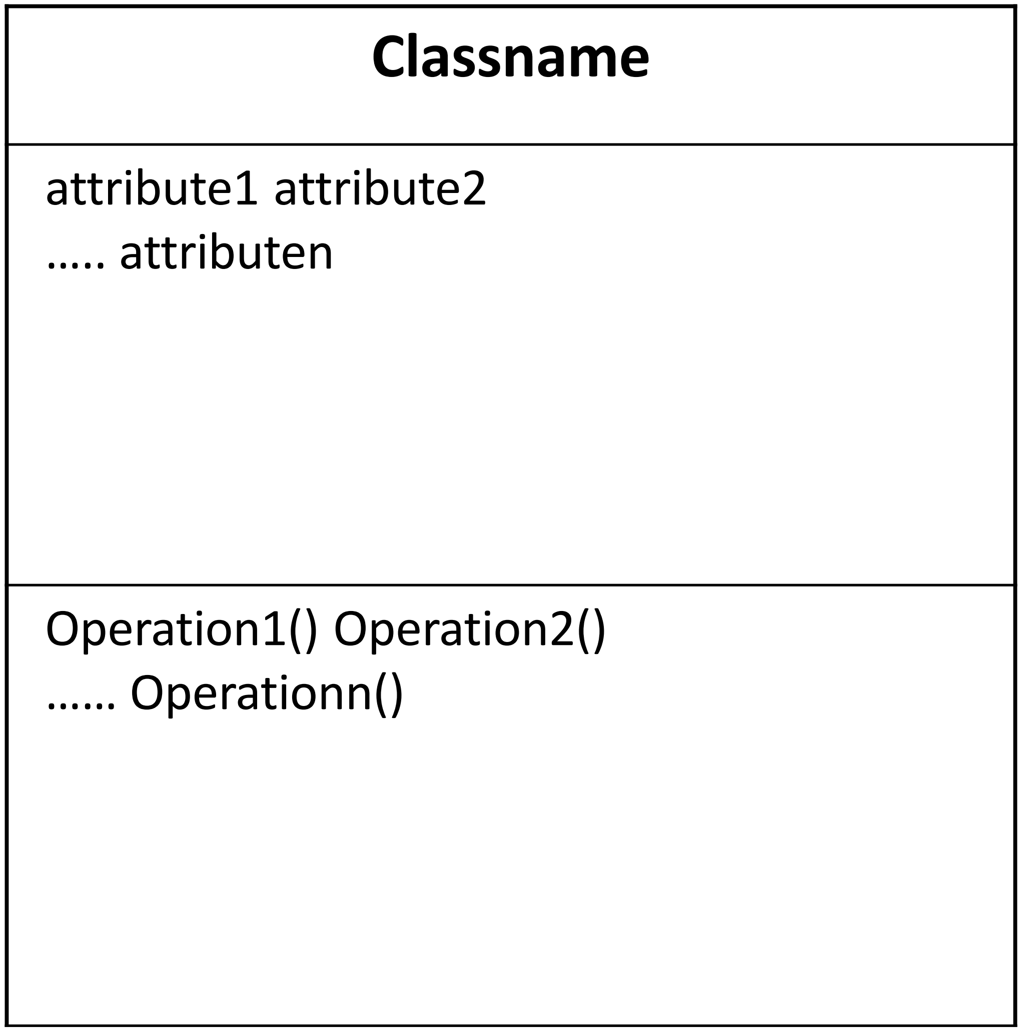
- The multiplicity of an atttribute indicates the number of attributes per instance of the class.
    - E.g. mark[3]:int
    - Here mark has three attributes mark[0],mark[1] , mark[2].
    - i.e. mark attribute can take 3 mark values
  - An attribute without square brackets [] must hold exactly one value.

Class diagram- OPERATIONS

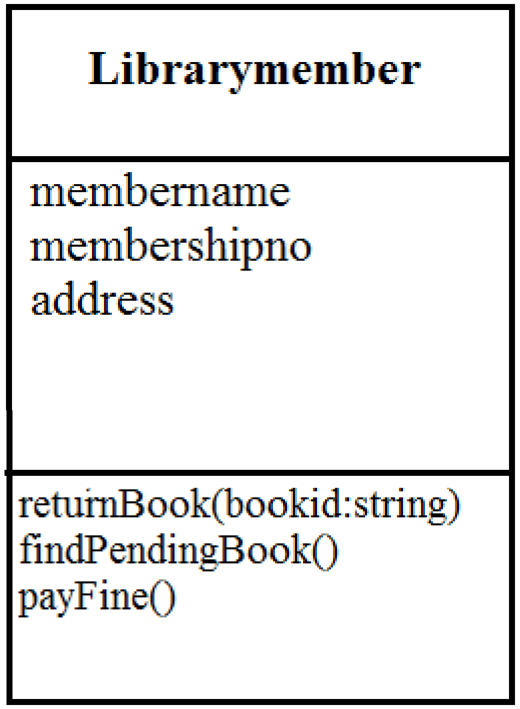
- Third compartment(optional)-**Operations-**
  - The operation names are **left justified**, in plain type, and always <u>begin with a lower case letter</u>.
  - Abstract operations are written in *italics*.
  - The ***parameters*** of a function may have following **kind** specified.
    - “in” - parameter is passed into the operation; (default)
    - “out”- parameter is only returned from the operation;
    - “inout” - parameter is used for passing data into the operation and getting result from the operation.
  - The ***return type*** of a method is shown after the colon at the end of the function signature.
  - The ***data type of parameters***(arguments) are shown after the colon following the parameter name
- E.g

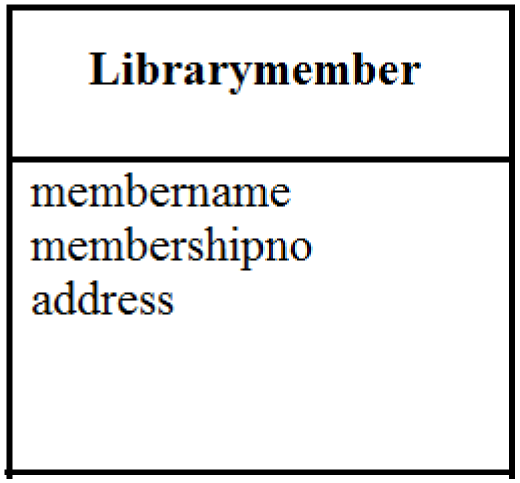


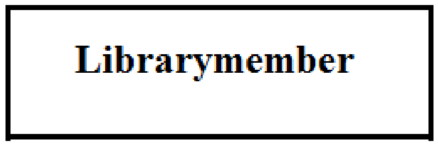
```
add(in a:int, inout b: int): int
```


Classname
attribute1 attribute2
….. attributen
Operation1() Operation2()
…… Operationn()

Class Diagram-examples





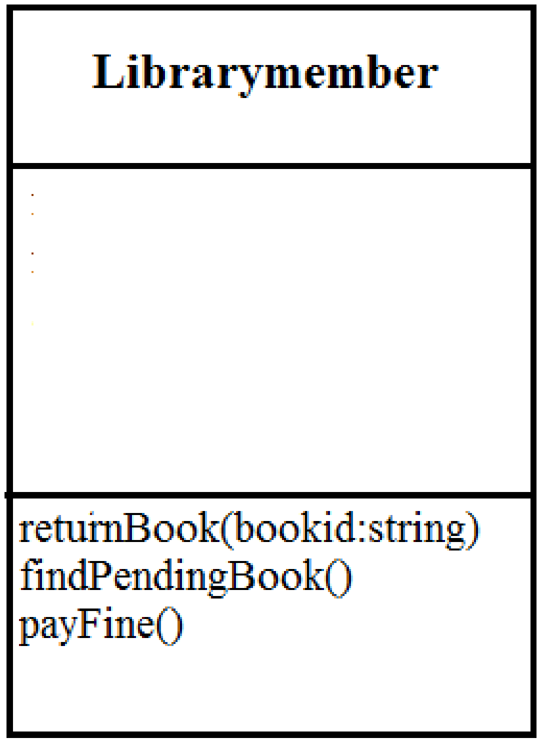


Class
No attribute
No operations

Class with attribute (no operations)

Class with attribute and operations

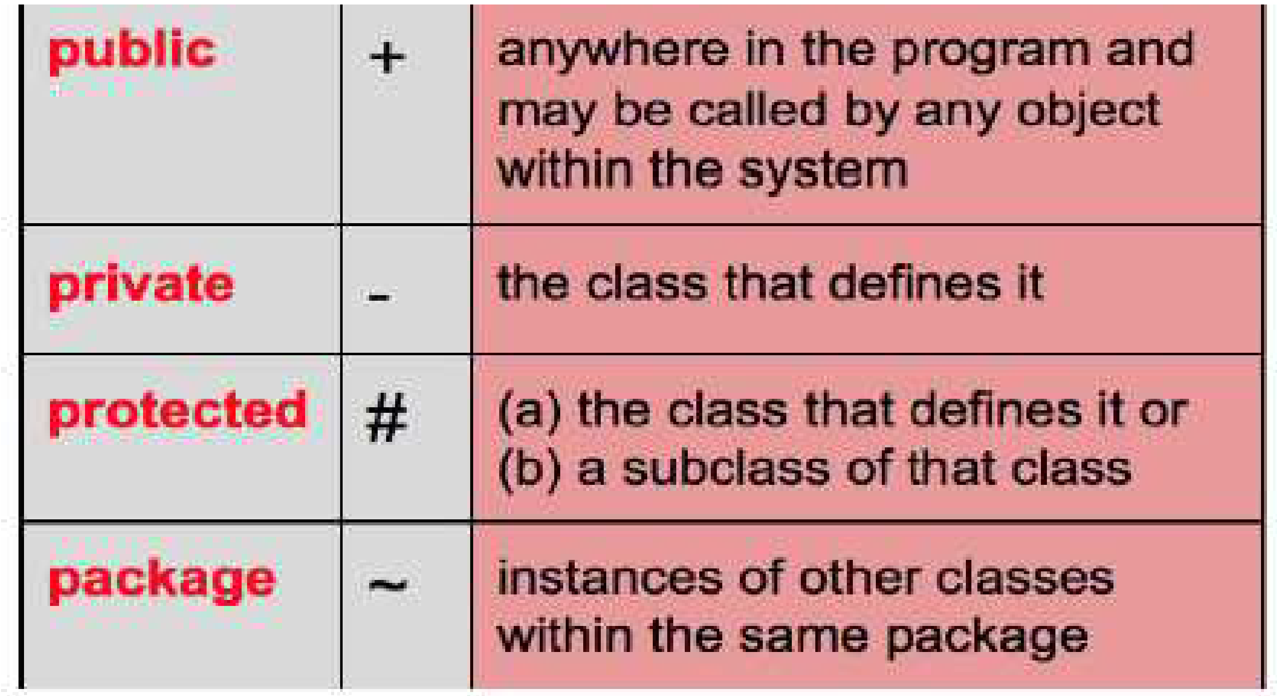
Class Diagram-examples



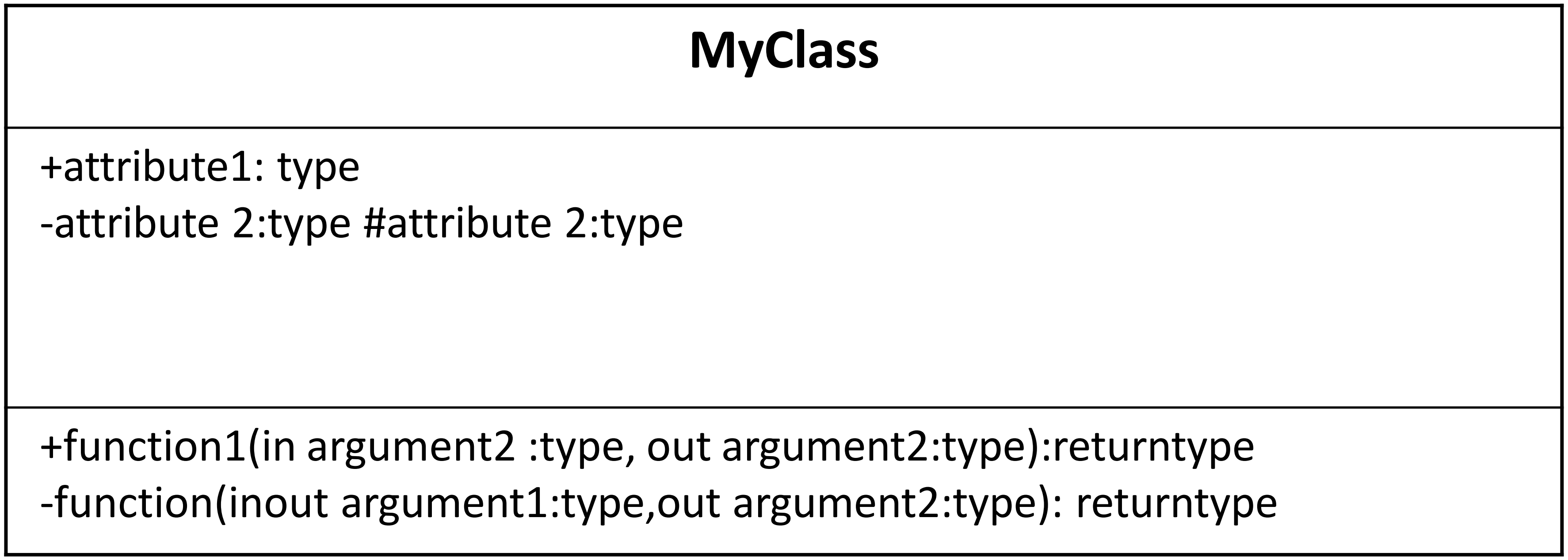
Class with operations but no no attributes

Class diagram-class visibility

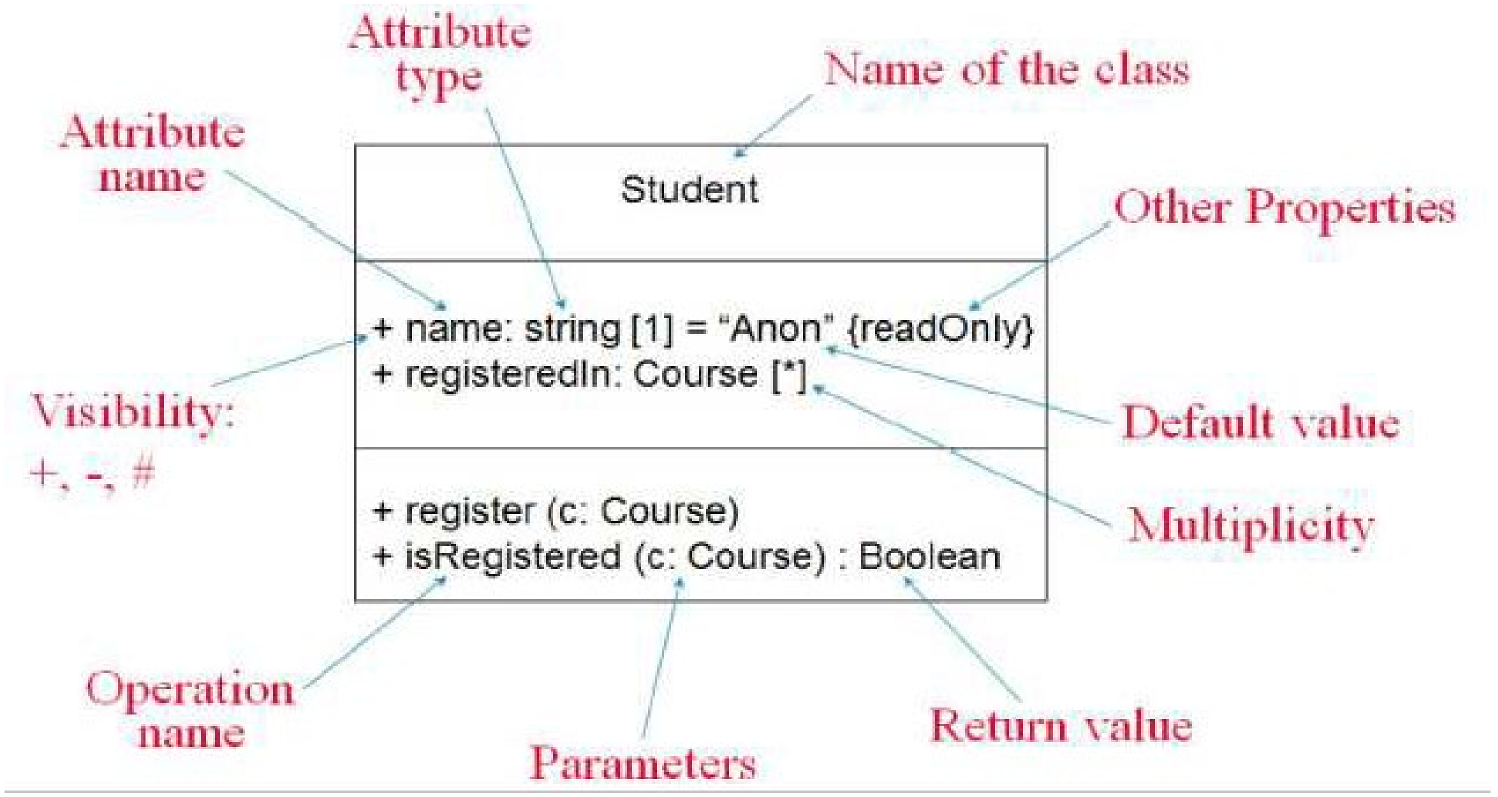
- To denote the visibility of the attribute and operation.



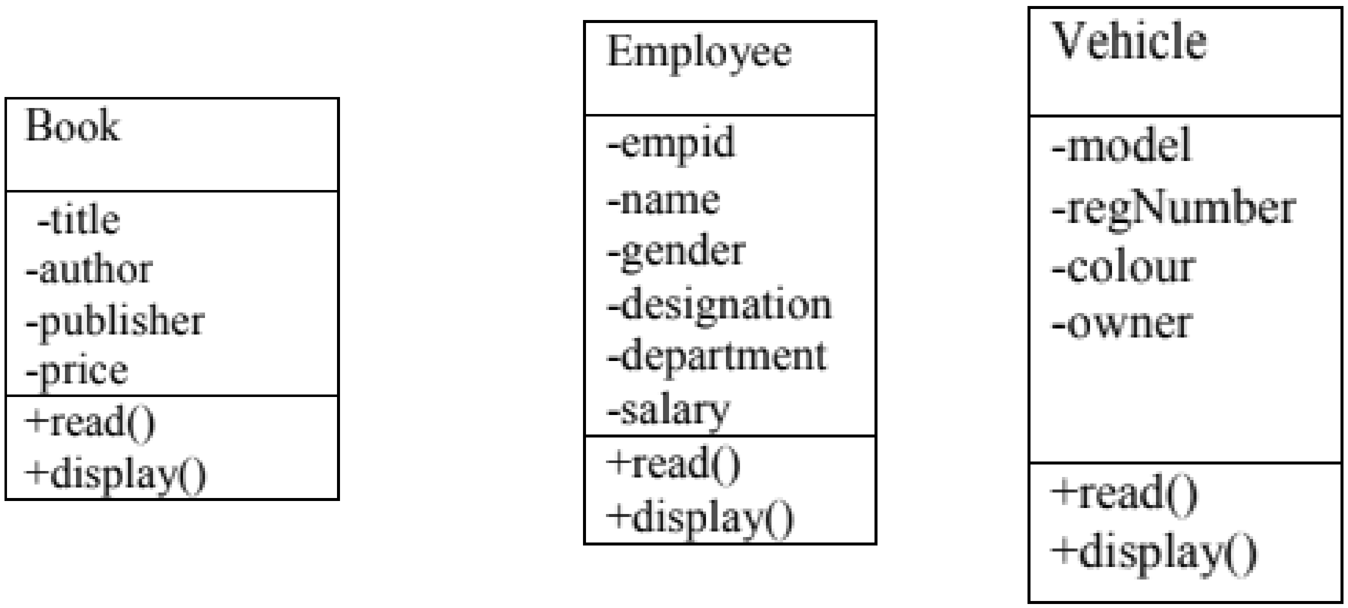
public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class
package	~	instances of other classes within the same package


MyClass
+attribute1: type
-attribute 2:type #attribute 2:type
+function1(in argument2 :type, out argument2:type):returntype
-function(inout argument1:type,out argument2:type): returntype

Class Diagram



- Represent the following entities using UML class diagram
- a.) Book b.) Employee c.) Vehicle



Association

- Association between two classes is represented by drawing a **straight line** between the concerned classes.
- The **name of the association** is written along side the association line.
- An **arrowhead** may be placed on the association line to *indicate the reading direction of the association*.

Association(contd.)

- On each side of the association relation, multiplicity can be noted
  - The **multiplicity** indicates how many instances of one class are associated with the other.
  - the **multiplicity** can be noted as an individual number or as a *value range*.
  - An **asterisk** * is used as a wild card and it means many (zero or more).

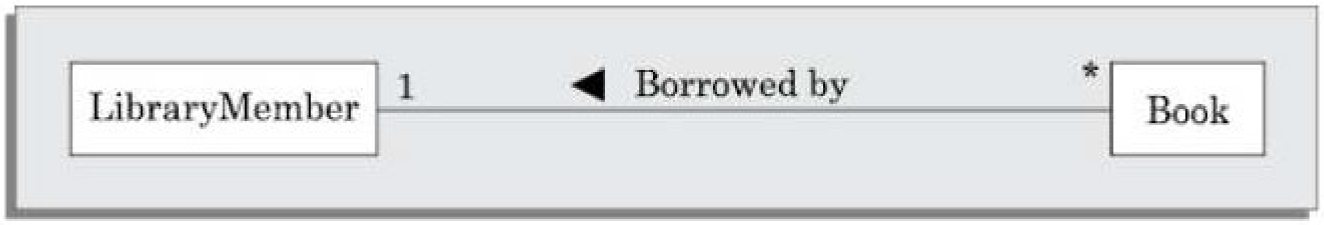
1 exactly one	* many

  - ***Value ranges*** of multiplicity are noted by specifying the minimum and maximum value, separated by two dots.

0..1 zero or 1	0..* zero or many
1..* 1 or many	2..5 minimum 2. maximum 5

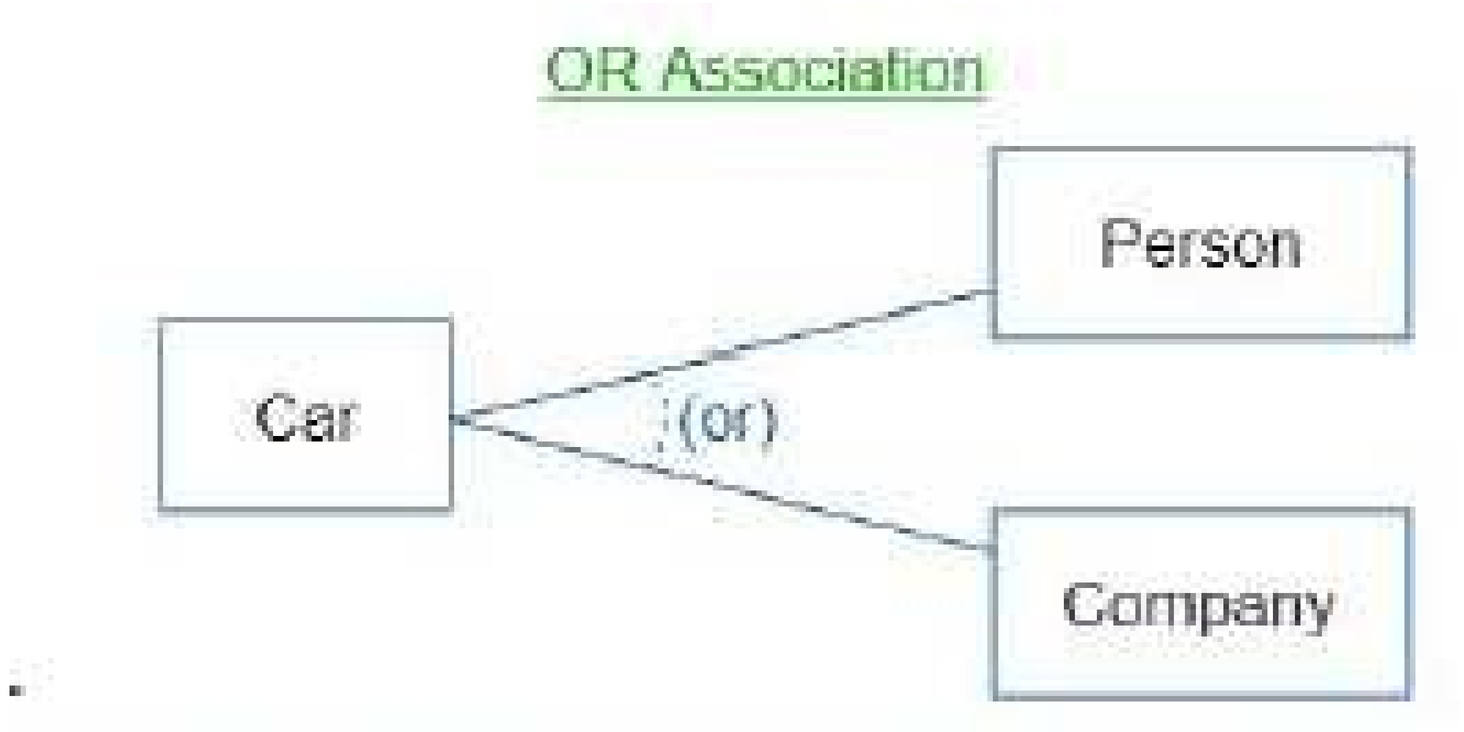
Prepared by Renetha J.B.

- "Many books may be borrowed by a LibraryMember".



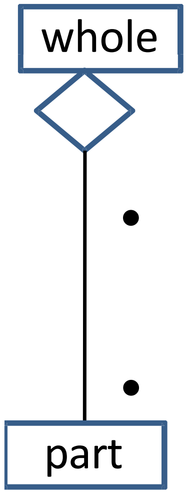
OR ASSOCIATION

- Car belongs to Person or Company

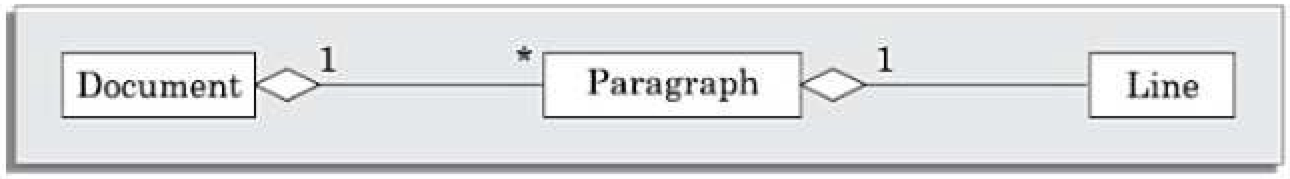


Aggregation

- Aggregation is a special type of association relation .
- Here classes are not only associated to each other, but a **whole-part** relationship exists between them.
- The parts are **NOT existence-dependent** on the whole.
  - E.g a book registeris an aggregation of book objects. Books can be added to the register required. Book can exist eventhough there is no Bookregister.
- Aggregation is represented by an **empty(unfilled) diamond symbol at** the aggregate end(**<u>whole</u>**) of a relationship.
- The aggregation relationship cannot be reflexive (i.e. recursive).
  - That is, an object cannot contain objects of the same class as itself.

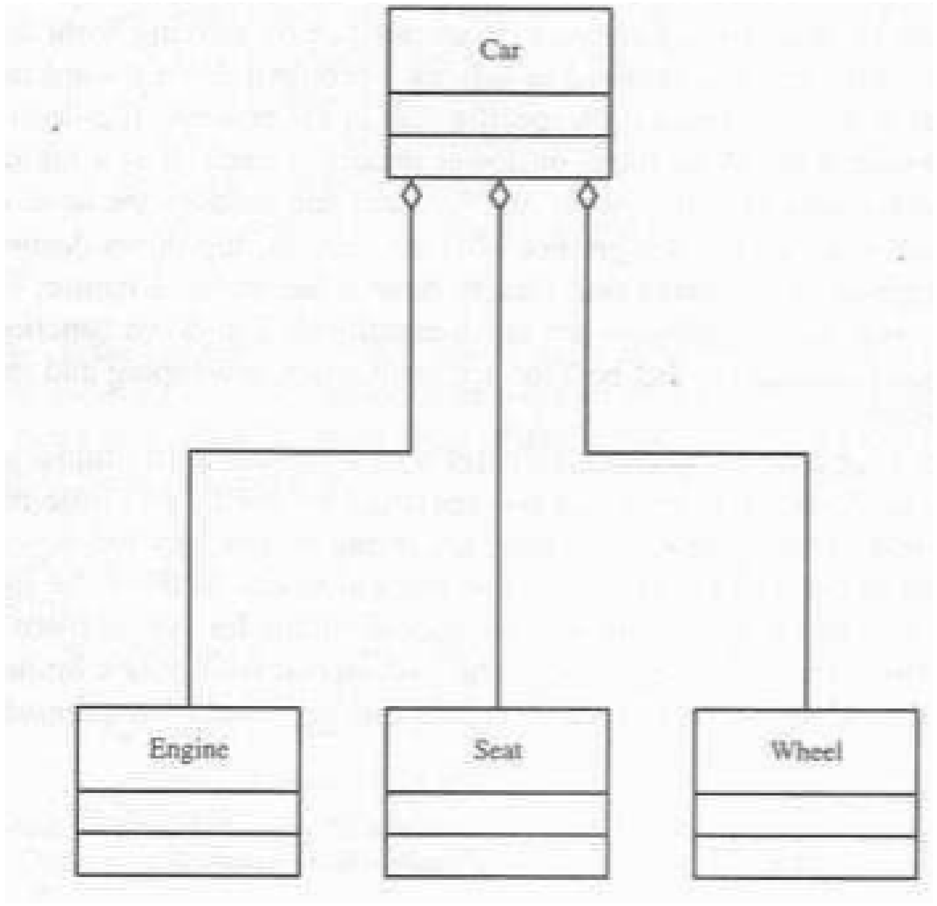


Aggregation



- Here document can be considered as an aggregation of paragraphs.
- Each paragraph can in turn be considered as aggregation of lines.
- Observe that the number 1 is annotated at the diamond end, and a * is annotated at the other end. This means that one document can have many paragraphs.
- If we wanted to indicate that a document consists of exactly 10 paragraphs, then we have to write 10

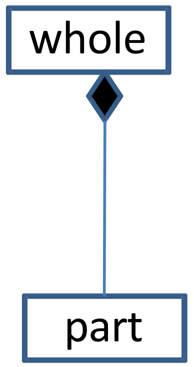
Aggregation

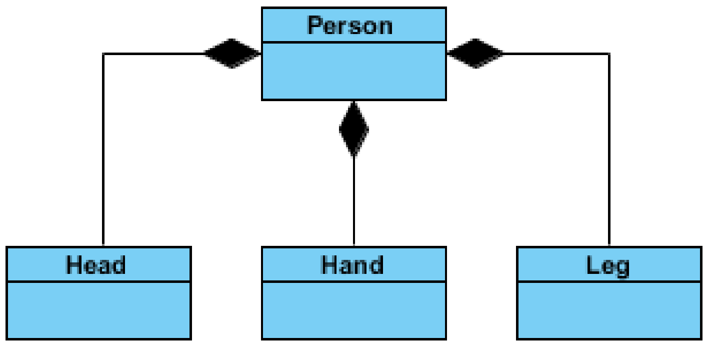


A car is an aggregation of engine, seat ,wheel. They are parts of car.
If car is damaged then engine seat and wheel can be used for another.

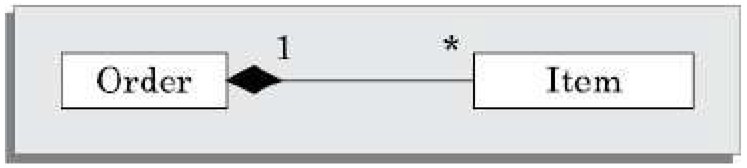
Composition

- Composition is a **stricter form of aggregation**, in which the parts are existence-dependent on the whole.
- If whole is lost then parts does not exist.
- Composition is represented as a **filled diamond** drawn at the composite-end.
- E.g. If man dies then his head, hand and leg also gets decayed.





E.g. If order is cancelled no significance for the item ordered.



Prepared by Renetha J.B.

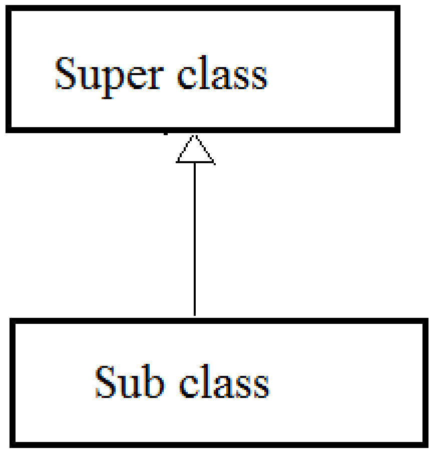
Aggregation versus Composition

- Both **aggregation and composition** represent **part/whole** relationships.
- When the **components can dynamically be added and removed** from the aggregate, then the relationship is **aggregation**
- If the **components CANNOT be dynamically added/delete**

  then the components are have the same life time as the composite. It is **composition**.
- Consider the example of an order consisting many order items.
  - If the order once placed, the items cannot be changed at all. In this case, the order is a **composition** of order items.
  - If order items can be changed (added, delete, and modified) after the order has been placed, then **aggregation** relation can be used to model it.

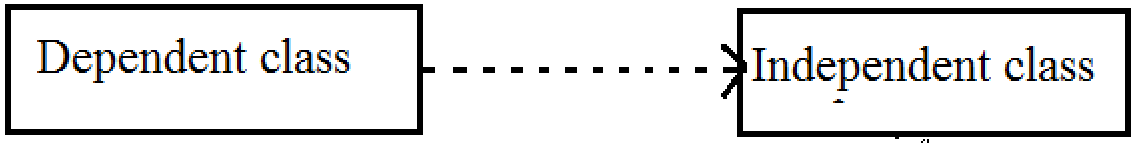
Inheritance

- A generalization (inheritance) helps to connect a subclass to its superclass.
  - A sub-class inherit properties from its superclass.
  - Class diagram allows inheriting from multiple superclasses
  - A **solid line with a hollow arrowhead** that point from the child to the parent class.

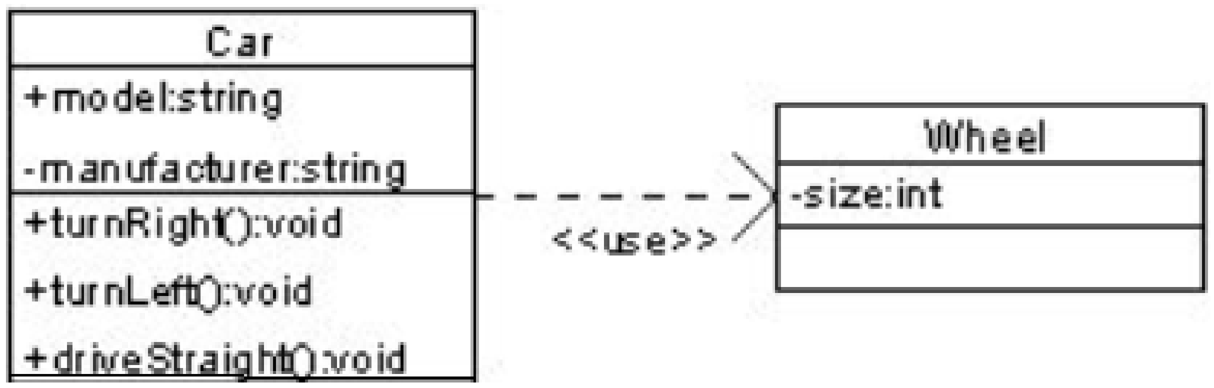


Dependency

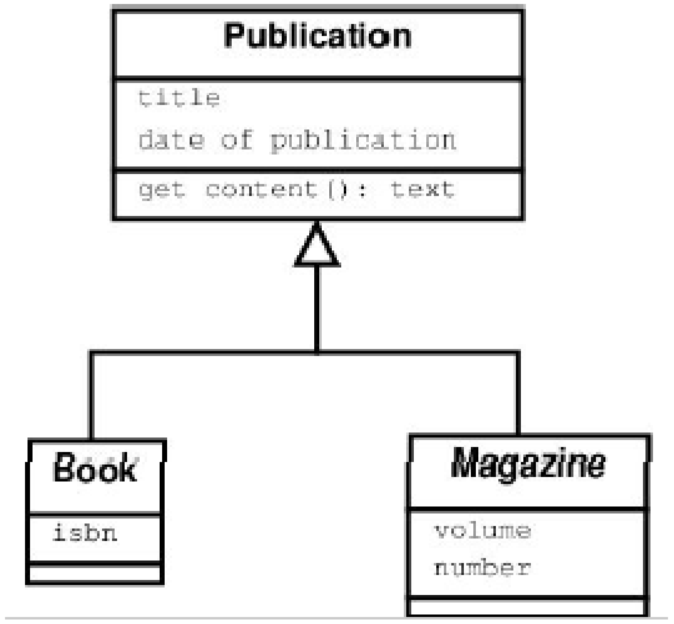
- A dependency means the relation between two or more classes in which a change in one may force changes in the other.
  - Dependency indicates that one class depends on another.
- A dependency relationship is shown as a **dotted line with open arrow** that is drawn from the dependent class to the independent class.

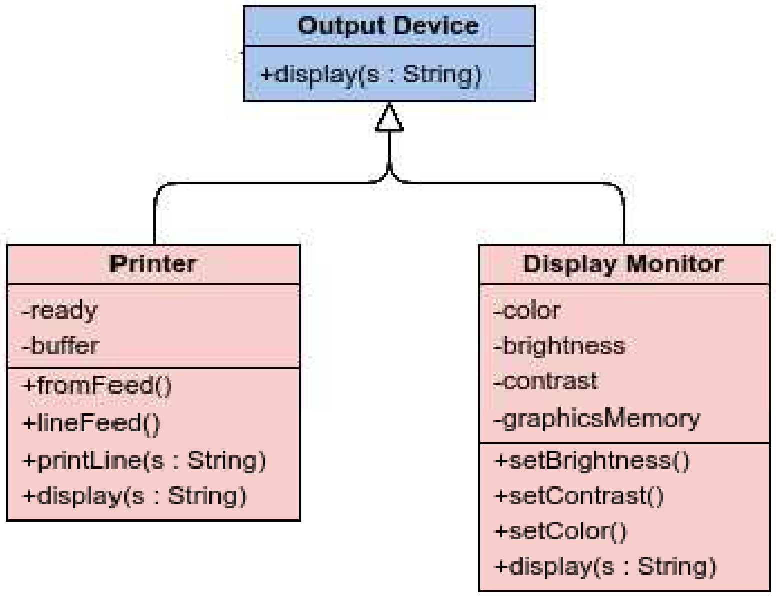


Dependency-Example



Inheritance(examples)



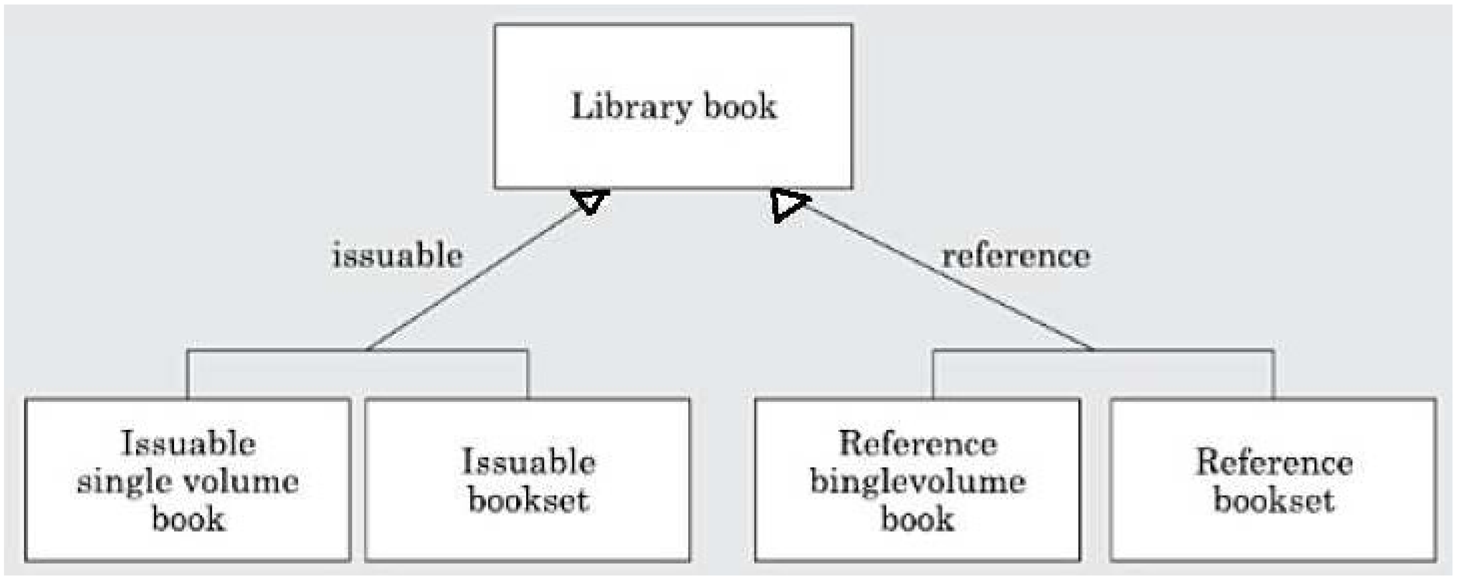


- Magazine is a publication.
- Book is a publication

Printer is an output device..
Display monitor is an output device.

Inheritance(contd.)

- The various subclasses of a superclass can then be differentiated by means of the discriminator. The set of subclasses of a class having the same discriminator is called a partition



Realization

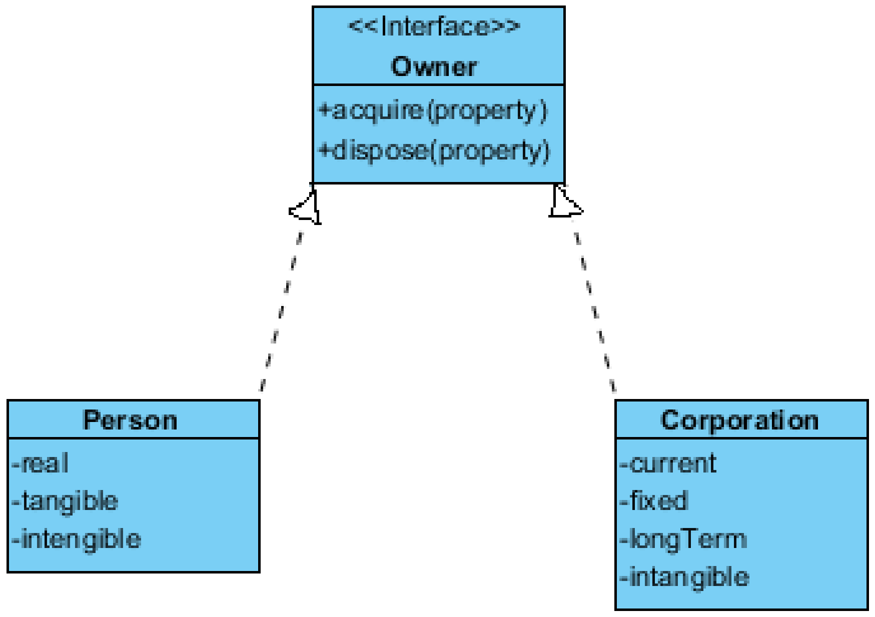
- **Realization (Implements)**
- Realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the **behavior** that the other model element (the supplier) **specifies**.
- It is represented as a **hollow triangle shape** on the interface end of the ***dashed*** **line.**

Realization

- **Interface**
- Use keyword **interface**
- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but **not how it does it**.
- Interfaces are syntactically similar to classes, but they **lack instance variables**.
- As a general rule, their **methods are declared without any body.**
- Alternate way to implement Multiple Inheritace in Java

Realization(contd.)

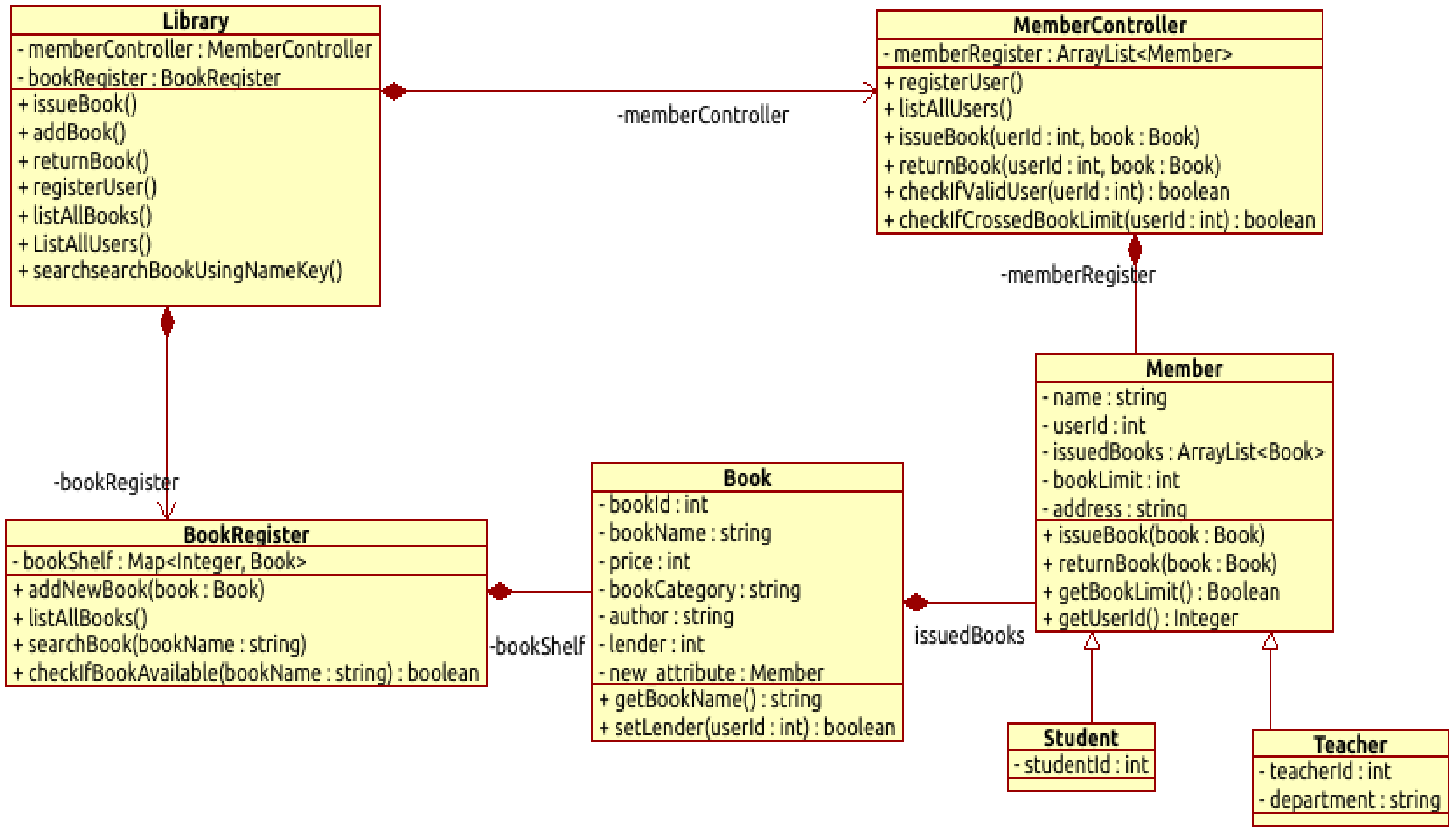
- E.g., the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, in very different ways.



Realization

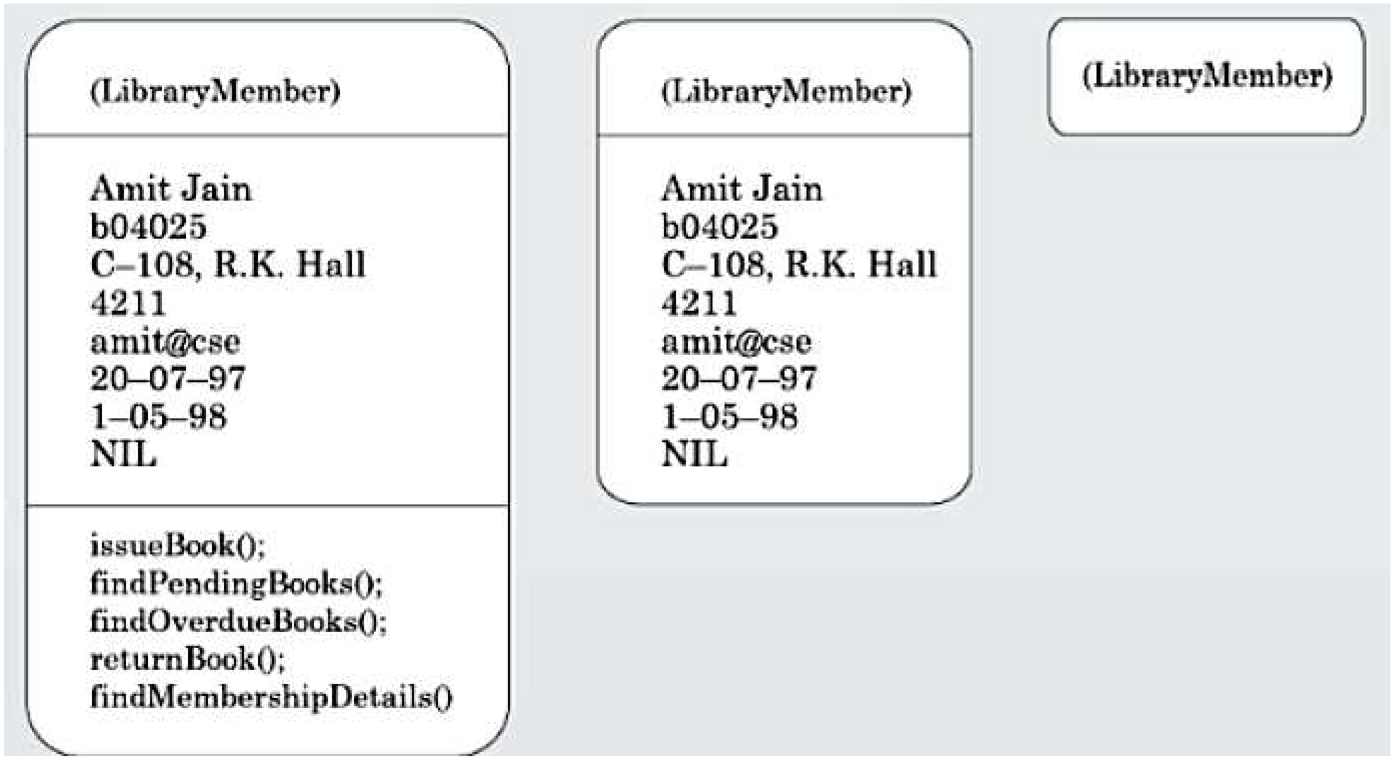
- **Implemening Interface**
  - To implement an interface, a class must provide the complete set of methods required by the interface.
  - However, each class is free to determine the details of its own implementation.
  - By providing the interface keyword, Java allows you to fully utilize the "**one interface, multiple methods**" aspect of ***polymorphism***.
  - Interfaces are designed to support dynamic method resolution at run time.
  - They disconnect the definition of a method or set of methods from the inheritance hierarchy
  - *Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the* ***method signatures*** *are compatible.*

Class Diagram – Library Management System

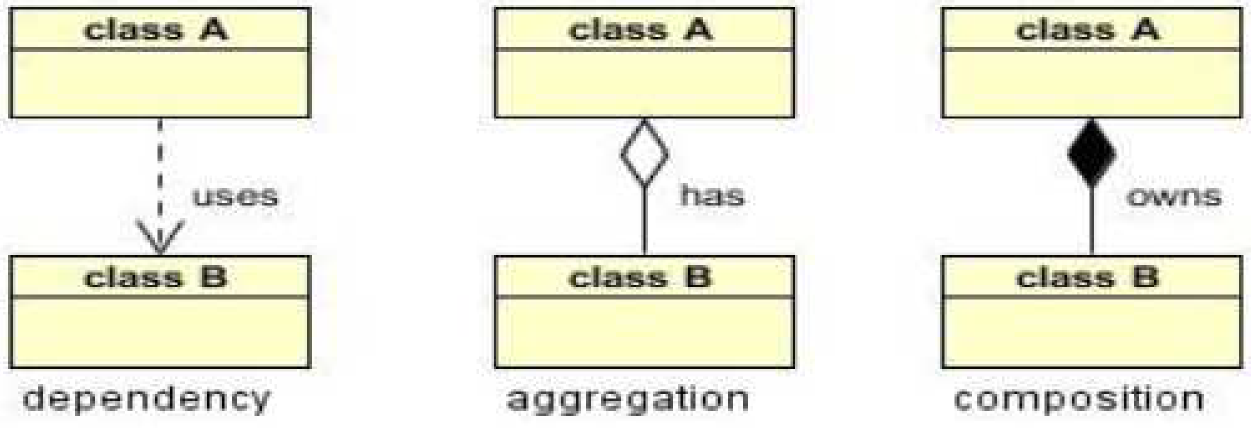


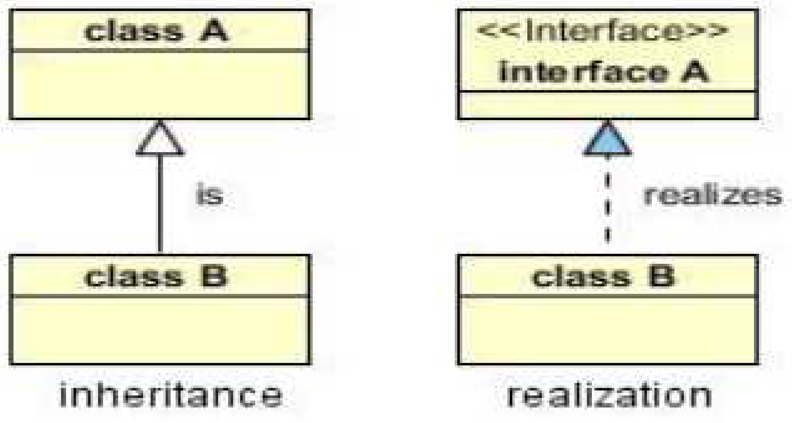
Object diagram

- Object diagrams shows the snapshot of the objects in a system at a point in time.
- Since it shows instances of classes, rather than the classes themselves, it is often called as an instance diagram.
- The objects are drawn using **rounded rectangles**



Object diagrams





Interaction Diagrams

- When a user **invokes one of the functions** supported by a system, the required behaviour is realised through the ***interaction of several objects*** in the system.
- Interaction diagrams, are models that describe how groups of objects interact among themselves through message passing to realise some behaviour.
- Each interaction diagram realises the **behaviour of a single use case.**
- For complex use cases, more than one interaction diagrams may be necessary to capture the behaviour,

Interaction Diagrams(Contd.)

- An interaction diagram shows a ***number of example objects and the messages*** ***<u>that are passed</u>*** <u>between the objects</u> within the use case.

- There are two kinds of interaction diagrams—
  - sequence diagrams
  - collaboration diagrams.

- Any one of these diagrams can be derived automatically from the other.

- These two actually portray different perspectives of behaviour of a system and different types of inferences can be drawn from them.

Interaction Diagram-Sequence diagram

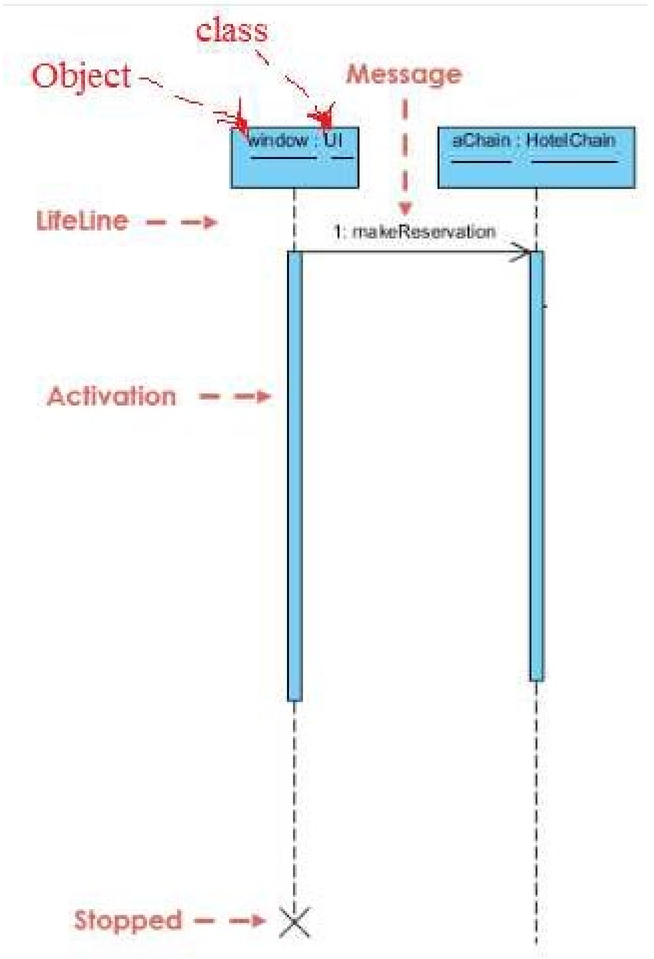
- A **sequence diagram** shows **interaction among objects** as a **two dimensional chart.** Sequence diagram shows only the behavioural aspects.
  - The chart is read from top to bottom.
  - The **objects** participating in the interaction are shown at the ***top of the chart as boxes attached to a vertical dashed line***.
  - Inside the box, the **name of the object** is written with a **colon** separating it from the **name of the class** and both the name of the object and the class are **underlined.**
    - This signifies that we are referring any arbitrary instance of the class.

Interaction Diagram-Sequence diagram(Contd.)

- An **object** appearing at the **top of the sequence diagram** signifies that the object existed even before the time the use case execution was initiated.
- The object should be shown at the appropriate place on the diagram where it is created.
- The **vertical dashed line** is called **the object's lifeline**.
  - **Absence of lifeline after some point** indicates that the object ***ceases to exist*** after that point in time, particular point of time.
  - If an object is **destroyed**, the lifeline of the object is **crossed** at that point and the lifeline for the object is not drawn beyond that point.
- A **rectangle** called the **activation symbol** is drawn on the lifeline of an object to indicate the points of time at which the **object is active**.

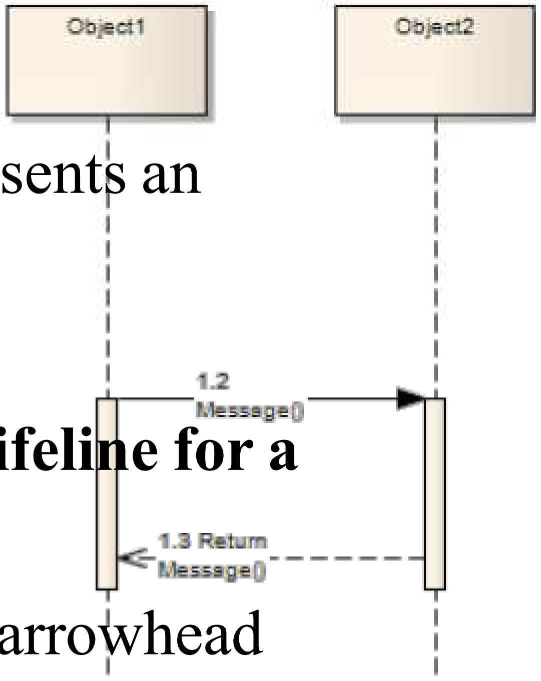
Interaction Diagram-Sequence diagram(Contd.)

- Each **message** is indicated as an arrow between the lifelines of two objects.
- The messages are shown in **chronological order** from the top to the bottom.
  - That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur.
- Each **message** is **labelled with the message name**. Some *control information* can also be included.
  - Two important types of *control information* are:
    - A **condition** (e.g., [invalid]) indicates that a message is sent, only if the condition is true.
    - An **iteration marker** shows that the message is sent many times to multiple receiver,


class
Object
Message
window : UI
aChain : HotelChain
LifeLine
1: makeReservation
Activation
Stopped

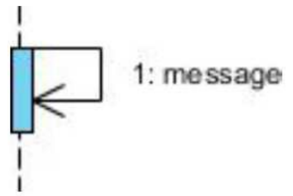
Messages

- A message defines a particular communication between Lifelines.
  - **1. Call Message**
    - Call message is a kind of message that represents an **invocation of operation of target lifeline.**
  - **2. Return Message**
    - Return message is a **reply from the target lifeline for a** previous message.
    - Represented with dashed line with an open arrowhead

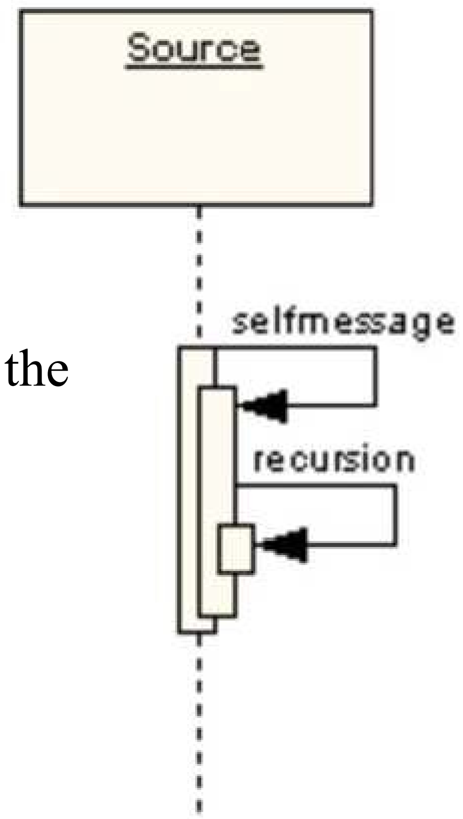


Messages(contd.)

- **3. Self Message**

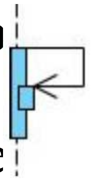


  – Self message is a kind of message that represents the **invocation** of message of the **same lifeline.**



- **4. Recursive Message**

  – A **self message sent for recursive purp**



  called a recursive messag

  e

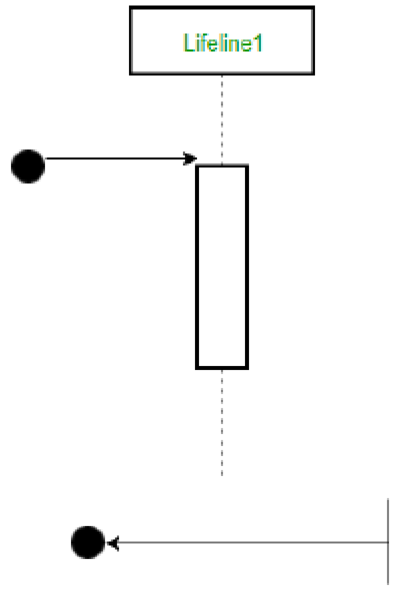
  – It's target points to an activation on top of the activation where the message was invoked from.

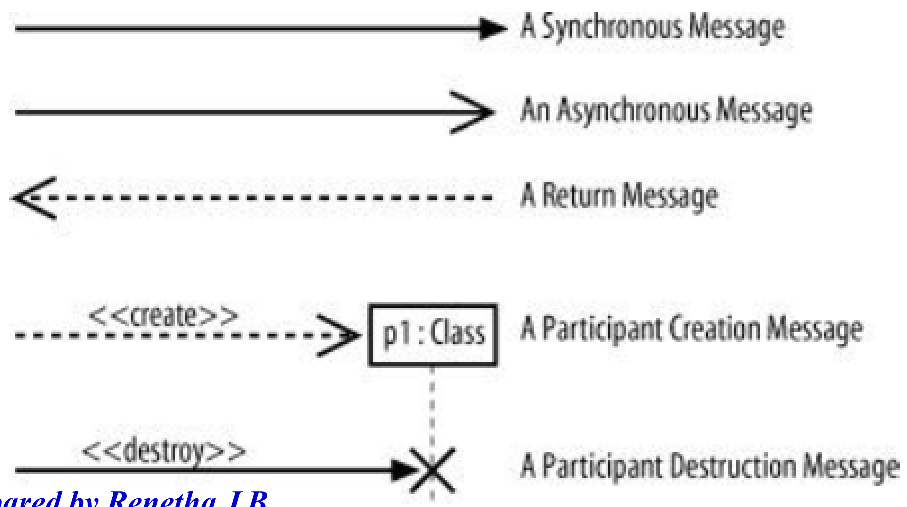
Messages(contd.)

- **5. Create Message / Constructor Message**
  - Create message is a kind of message that represents the creation of (target) lifeline.
  - It **creates the message receiver**
- **6. Destroy Message / Destructor Message**
  - Destroy message is a kind of message that represents the request of destroying the target lifeline.
  - It **destroys the message receiver.**
- **7. Duration Message / non-instantaneous message**
  - Duration message **shows the time delay between** the send time and receive time of the message. (eg, network communication delay)

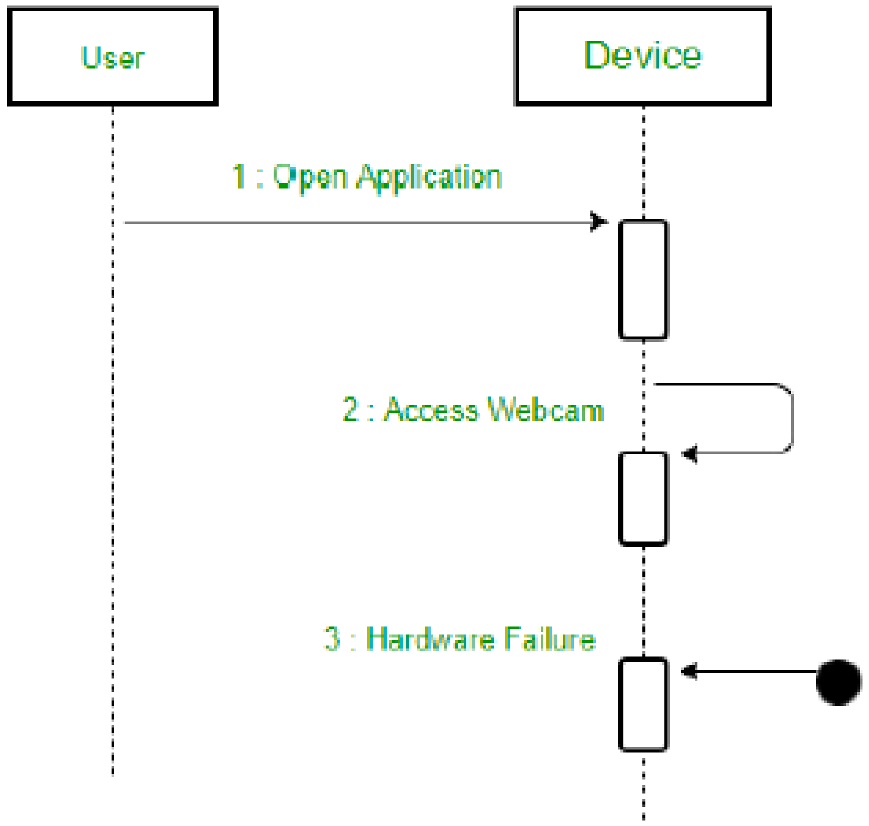
Messages(contd.)

- **8. Found message - Unknown sender**
  - A Found message is used to represent a scenario where an unknown source sends the message.
- **9. Lost message - Unknown receiver**
  - A Lost message is used to represent a scenario where the recipient is not known to the system.

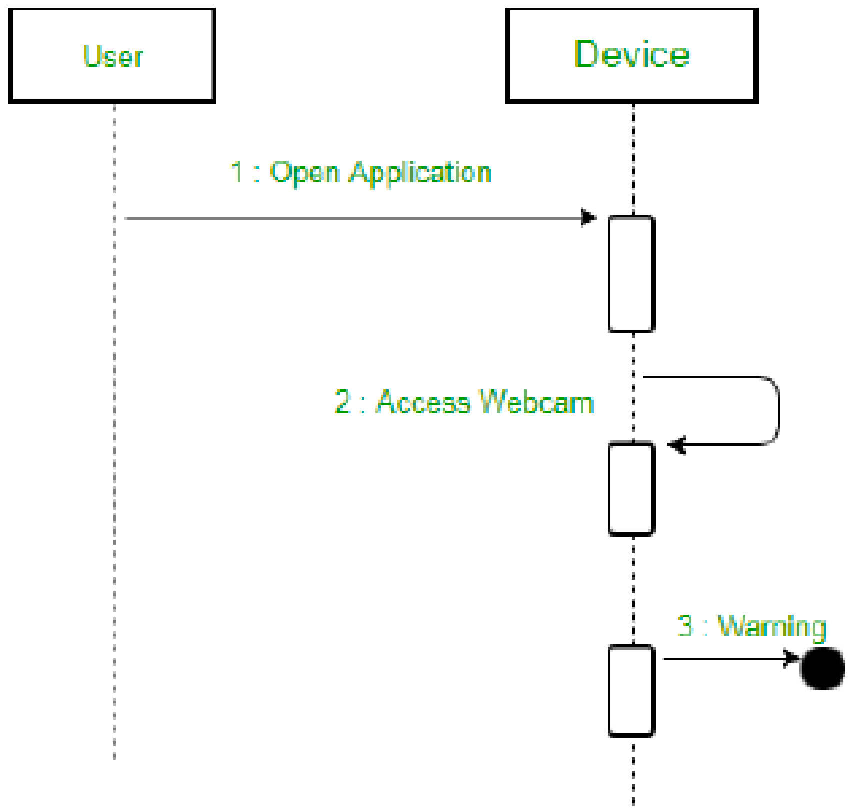




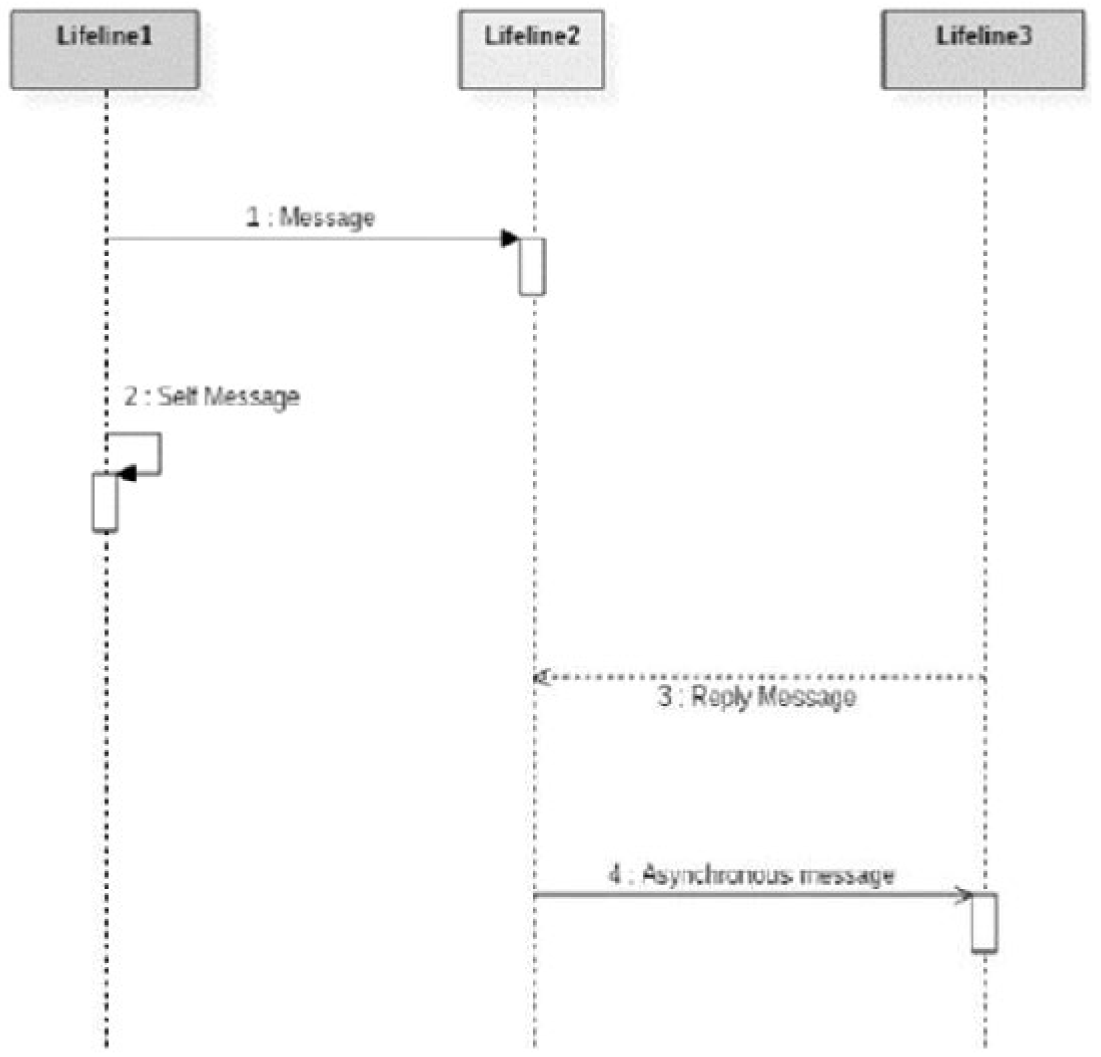
- Example for **found** message : unknown sender



- Example for **lost** message: Unknown receiver

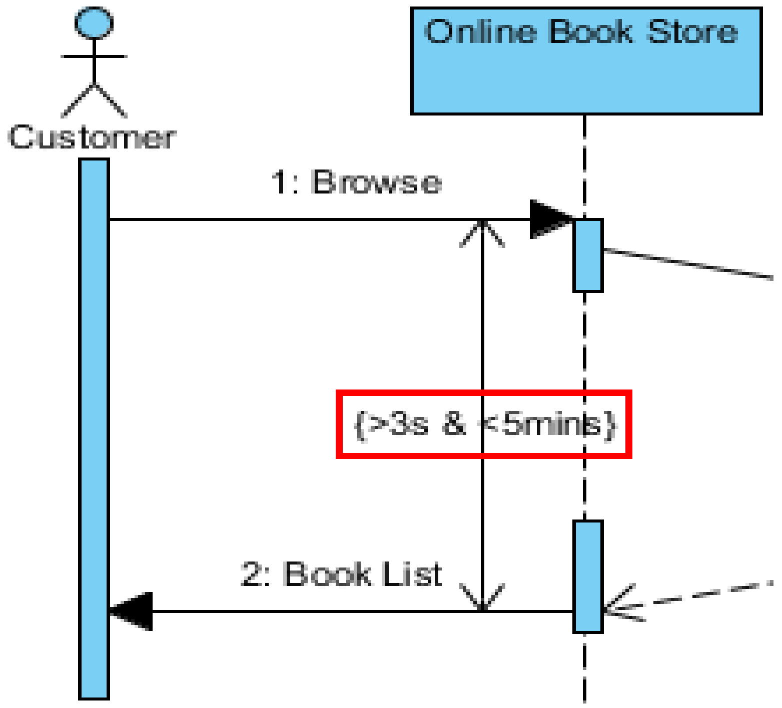


A message is lost

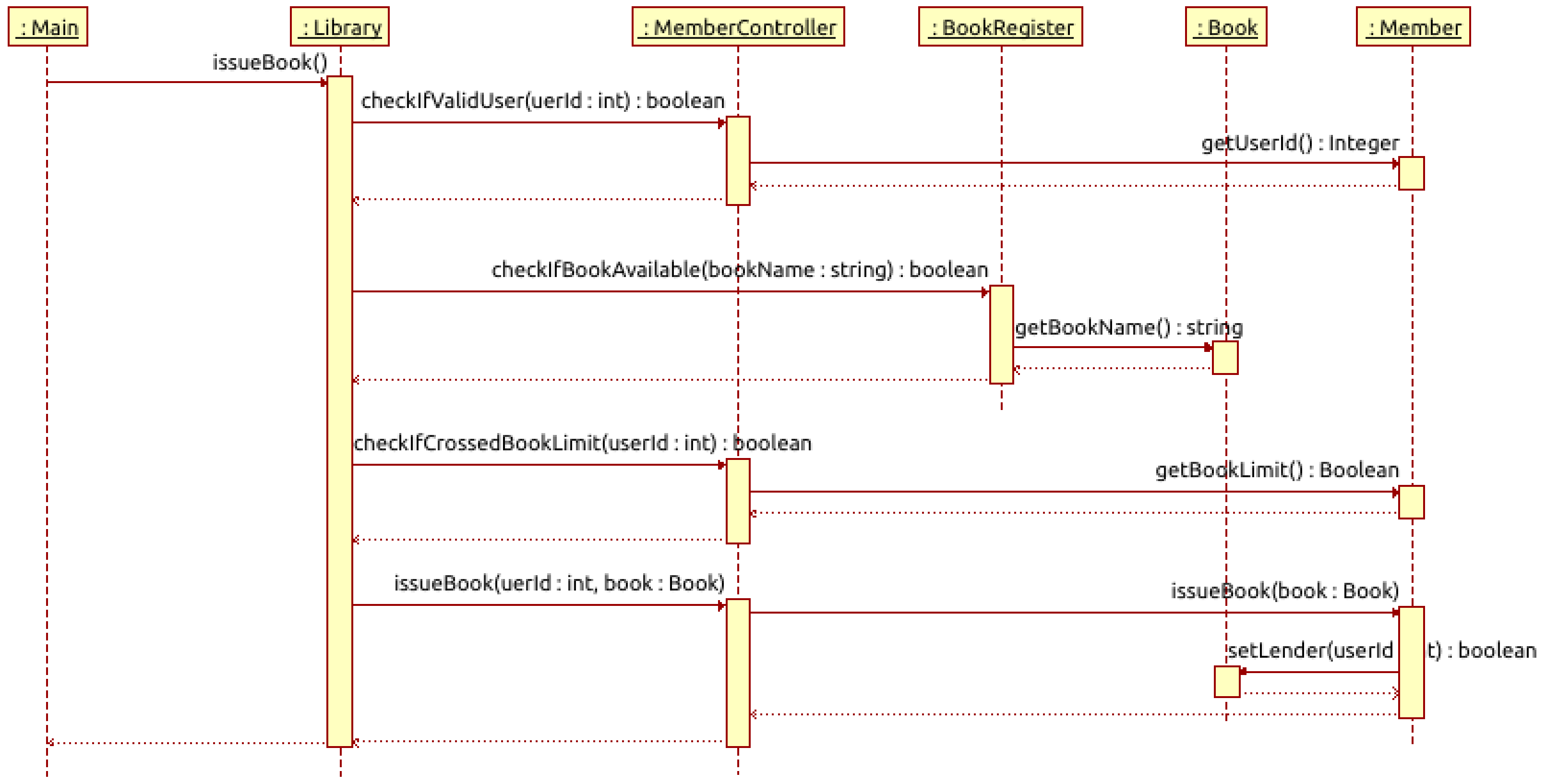


Source: https://www.geeksforgeeks.org/

Duration Message

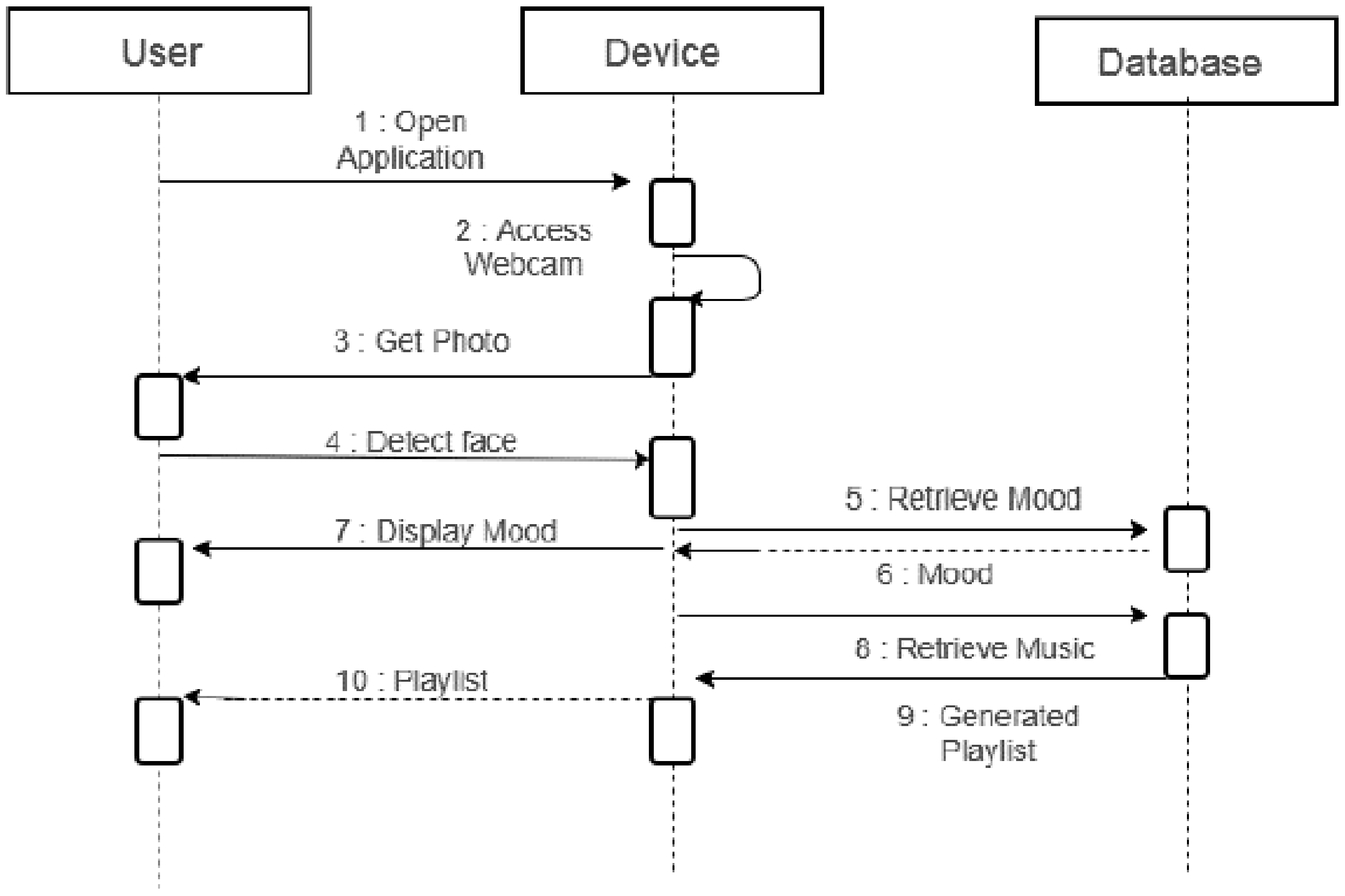


Sequence diagram for the issue book use case


: Main
: Library
: MemberController
: BookRegister
: Book
: Member
issueBook()
checkIfValidUser(uerId : int) : boolean
getUserId() : Integer
checkIfBookAvailable(bookName : string) : boolean
getBookName() : string
checkIfCrossedBookLimit(userId : int) : boolean
getBookLimit() : Boolean
issueBook(uerId : int, book : Book)
issueBook(book : Book)
setLender(userId : int) : boolean

Prepared by Renetha J.B.

Sequence diagram-E.g.



Emotion based music player

Interaction diagram-Sequence diagram(contd.)

- **Advantages**
  - Sequence diagrams are **easier to maintain and easier to generate.**
  - Represent the **low level details of a UML use case.**
  - Model the **logic of a complex procedure, function, or operation.**
  - Shows the **interaction between the objectsand components to complete a** process.

Interaction diagram-Sequence diagram(contd.)

- **Disadvantages**
  - Can be **complex** when too many lifelines are involved in the system.
  - If the <u>order of message sequence is changed</u>, then **incorrect results** are produced.
  - The type of message decides the type of sequence inside the diagram
  - Each <u>sequence</u> needs to be <u>represented using different message numbering schemes,</u> which can be a little **complex** in large systems

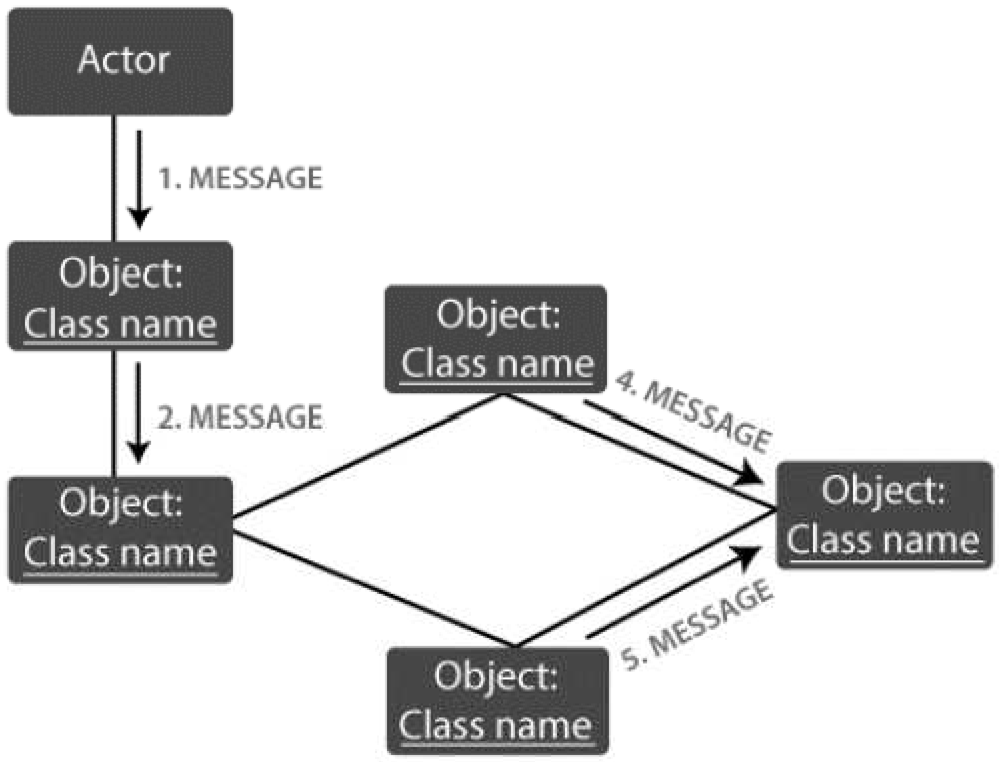
Interaction diagram-Collaboration diagram

- A collaboration diagram **shows** both structural and behavioural aspects explicitly.
- Each **object** is also called a **collaborator**.
  - The **behavioural** aspect is described by the set of **messages exchanged** among the different collaborators.
  - The **structural** aspect of a collaboration diagram consists of **objects and links** among them indicating association.

Interaction diagram-Collaboration diagram (contd.)

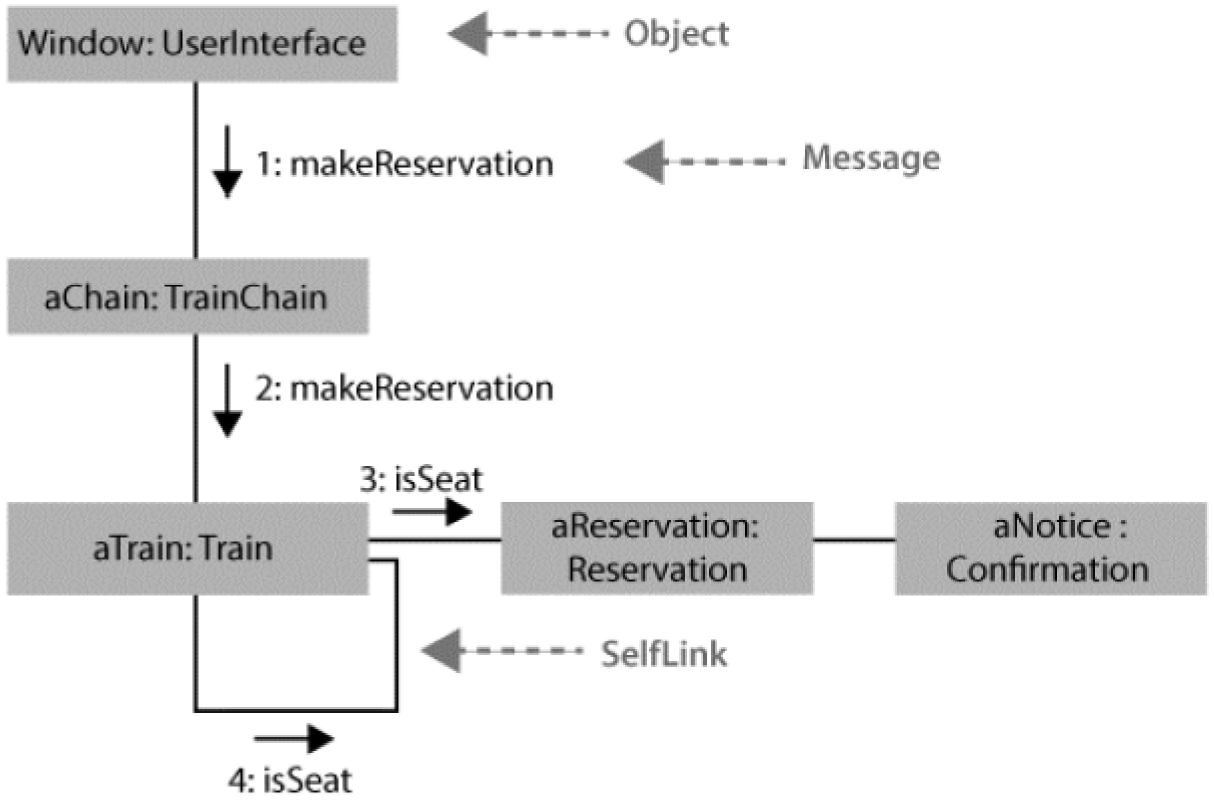
- The **link** between objects is shown as a solid line and can be used to **send messages** between two objects.
- The **message** is shown as a **labelled arrow** placed near the link.
- Messages are prefixed with sequence numbers because they are the only way to describe the *relative sequencing* of the messages in this diagram.
- Collaboration diagrams helps us to **determine which classes are associated with which other classes.**

Collaboration diagram(contd.)

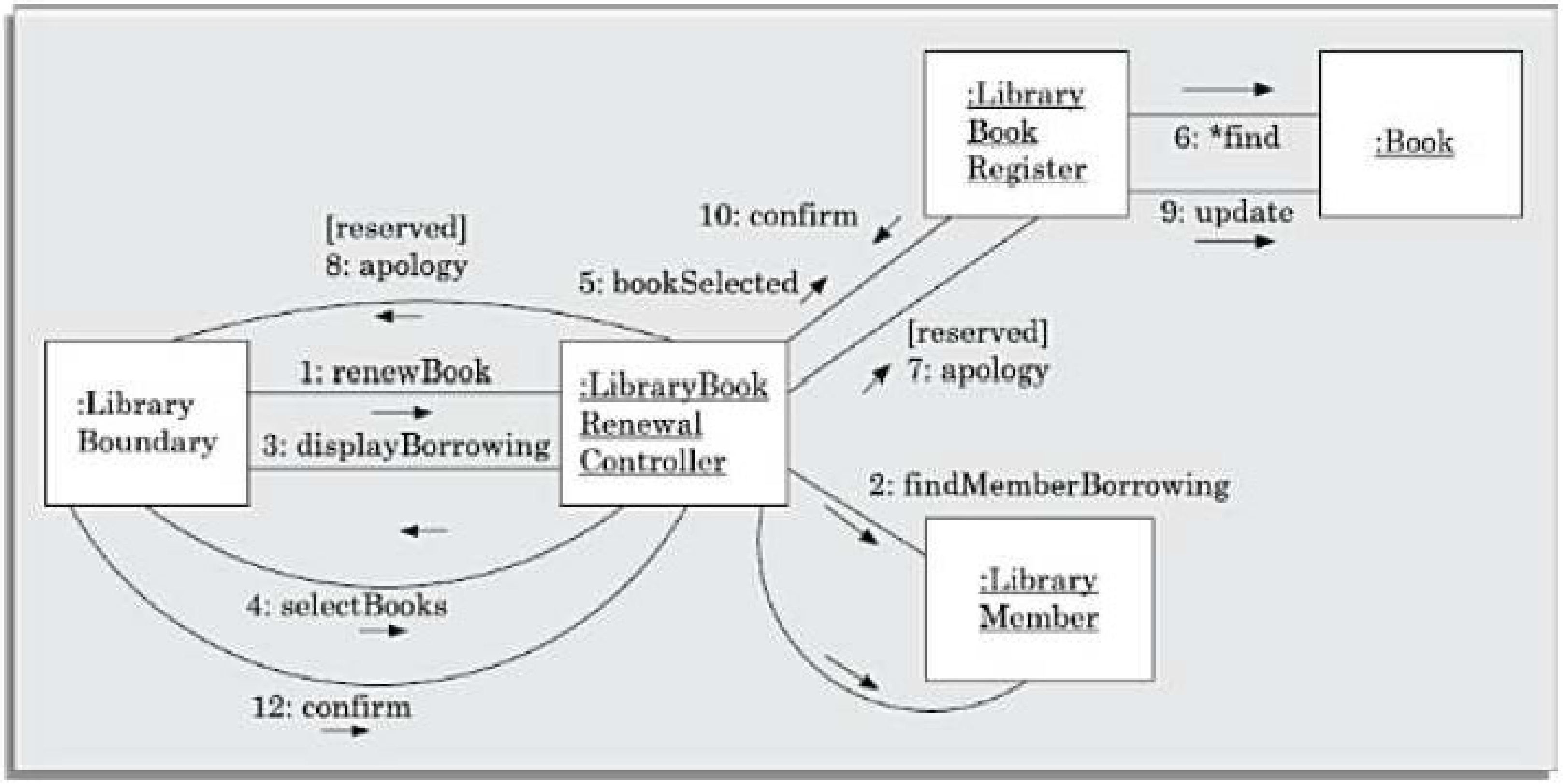


Source: https://www.javatpoint.com

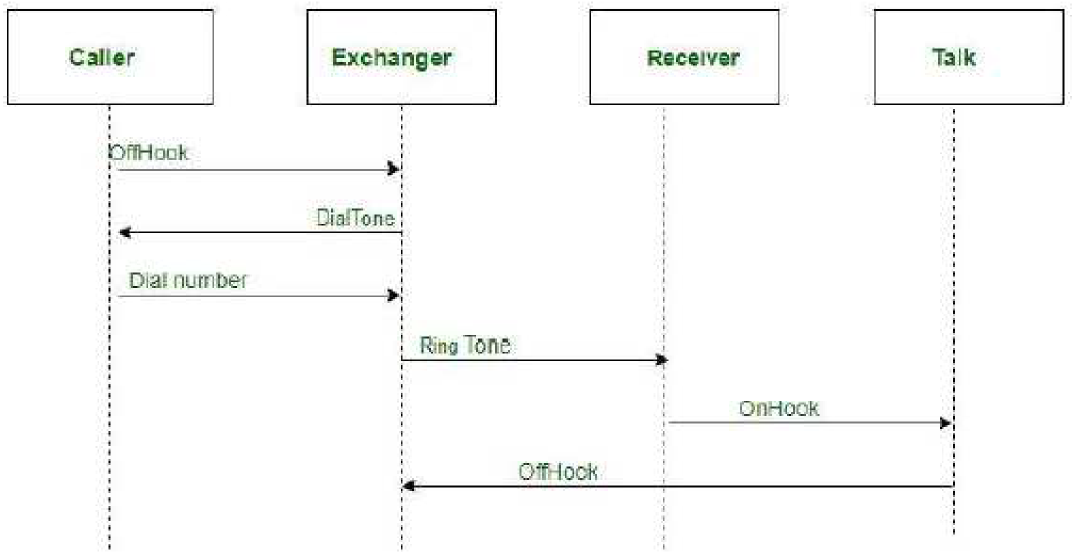
Collaboration diagram E.g.



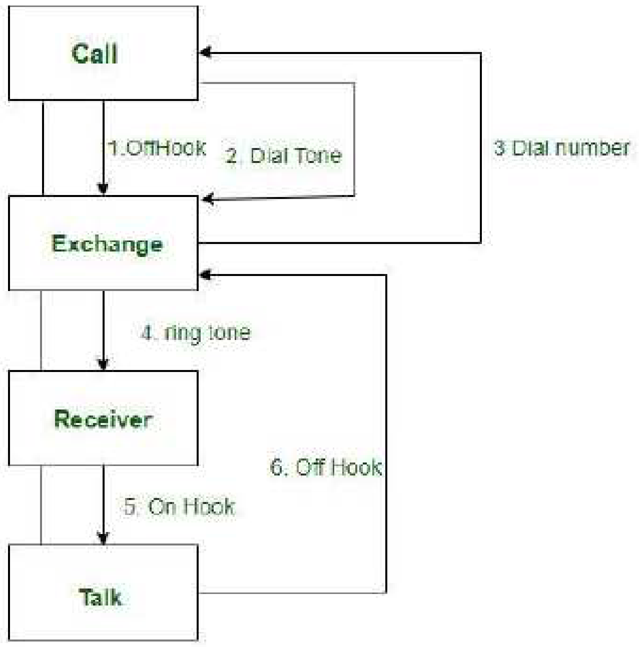
Collaboration diagram E.g.



Difference between sequence and collaboration diagrams



Sequence diagram



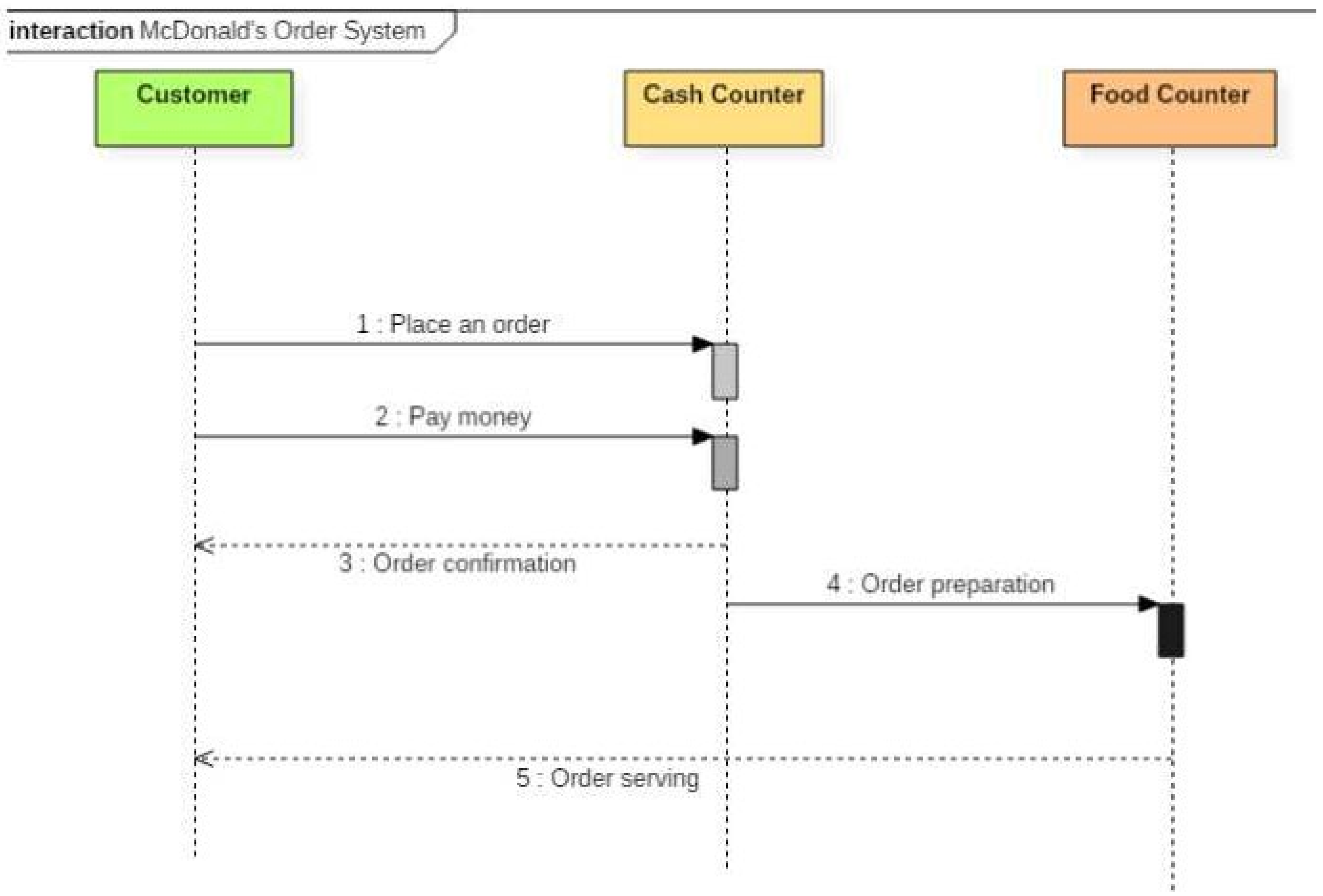
Collaboration diagram

Interaction diagram-Collaboration diagram (contd.)

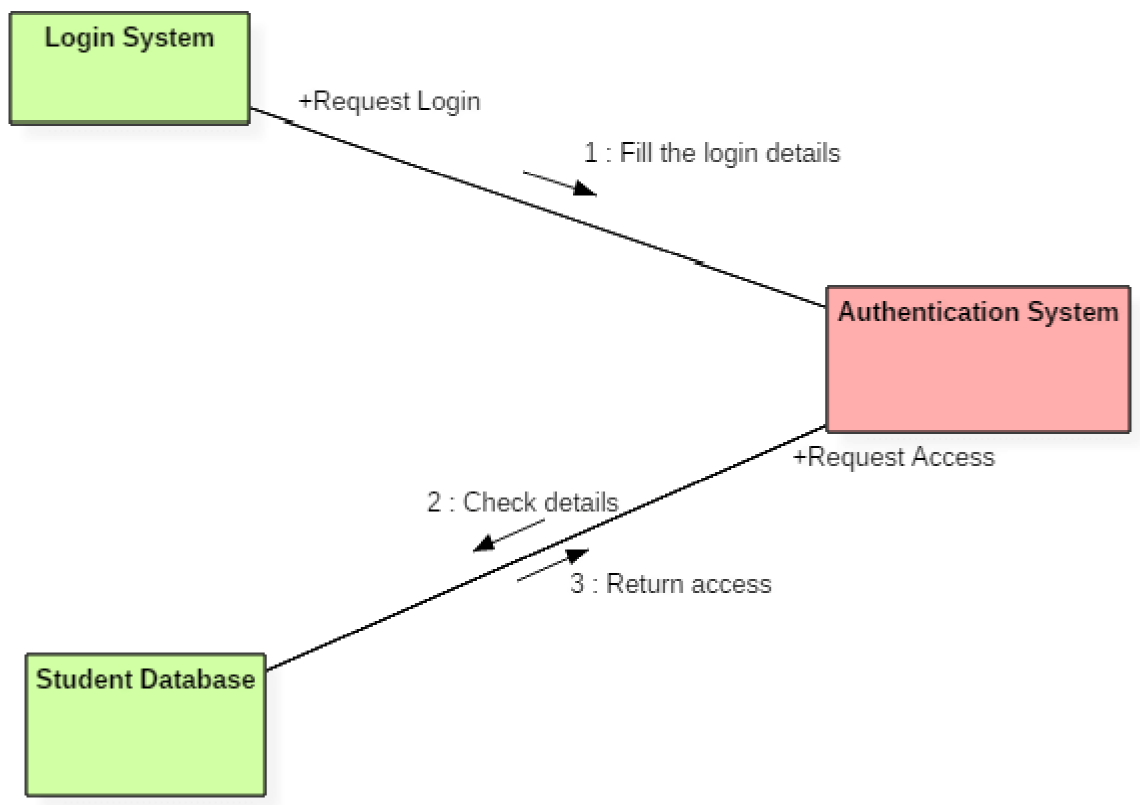
Sr. No.	Key	Sequence Diagram	Collaboration diagram
1	Definition	Sequence diagram is the diagram in which main representation is of the sequence of messages flowing from one object to another; also main emphasis is on representing that how the messages/events are exchanged between objects and in what time-order.	On other hand, Collaboration diagram is a diagram in which main representation is of how one object is connected to another implementing the logic behind these objects with the use of conditional structures, loops, concurrency, etc.
2	Main focus	Sequence diagram mainly focuses to represent interaction between different objects by pictorial representation of the message flow from one object to another object. It is time ordered that means exact interactions between objects is represented step by step.	On other hand Collaboration diagram focus to represent the structural organization of the system and the messages that are sent and received.
3	Type	As Sequence diagram models the sequential logic, ordering of messages with respect to time so it is categorised as Dynamic modelling diagram.	On other hand Collaboration diagram mainly represent organization of system so it is not classified as Dynamic modelling diagram.
4	UseCase	Sequence diagram as already mentioned is used to describe the behaviour of several objects in a particular single use case with implementation of all possible logical conditions and flows.	However on other hand Collaboration diagrams is used to describe the general organization of system for several objects in several use cases.

Sequence diagram – ordering system

The following sequence diagram example represents McDonald's ordering system:



Collaboration diagram- eg Student login management



Reference

- Rajib Mall, **Fundamentals of Software Engineering, 4th edition, PHI, 2014**

CS205 Object Oriented Programming in Java

Module 1 - **Introduction**

(Activity diagram, State chart diagram)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

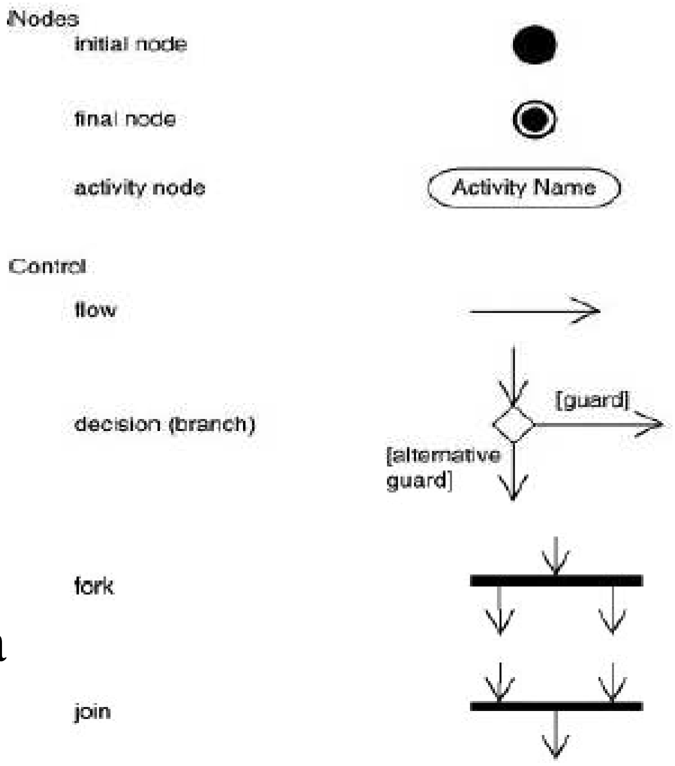
- Introduction:
  - ✓UML diagrams
    - ✓ Activity diagram
    - ✓State chart diagram

Activity diagram

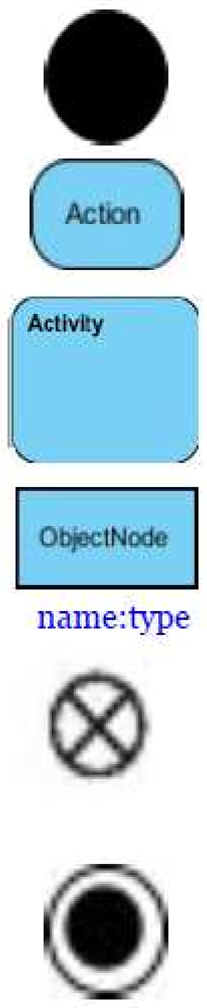
- The activity diagram focuses on representing various **activities** or **chunks of processing** and their **sequence of activation**.
  - The activities in general may not correspond to the methods of classes. **Activity diagram** is flow of ***activities without trigger*** (event) mechanism.(**not event driven**)
- An activity is a **state** with an internal action and one or more outgoing transitions. It is automatically followed by the termination of the internal activity.
- If an activity has more than one outgoing transitions, then exact situation under which each is executed must be identified through appropriate conditions.

Basic components of activity diagram

- **1. Nodes**
  - **Action:** A single step in the activity wherein the users or software perform a given task.
  - **Start node:** Symbolizes the **beginning** of the activity.
    - The start node is represented by **black circle**.
  - **End node:** Represents the **final step** in the activity.
    - The end node is represented by a **outlined black circle**.
- **2. Control flows:** also called **connectors.**
  - They show the control flow between steps in the diagram.
- **3. Partitions (optional) –**
  - to organize the nodes in the activity diagram.



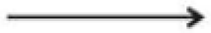
Nodes- notations



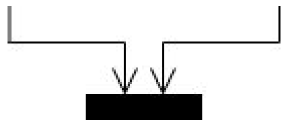
- Start symbol - Portrays the beginning of a set of actions or activities.
- Action symbol - A task to be performed
- Activity symbol - Indicates the set of actions that make up a modeled process. Includes short descriptions within the shape.
- Object Node - A node that is used to define object flow in an activity
- Flow final symbol - Represents the end of a specific process flow.
  - Does not represent the end of all flows in an activity.
  - No effect on parallel flows.
- End symbol - Stop all control flows and object flows in an activity.
  - Represented by an outlined black circle

Control flow-notations

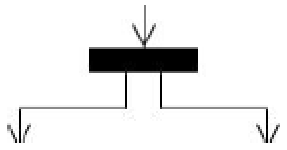
- **Action Flow** or **Control flow** - referred to as paths and edges. They are used to show the transition from one activity state to another.



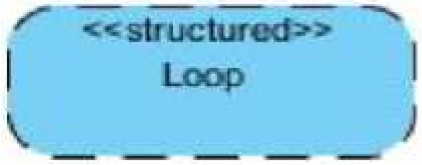
- **Joint symbol / Synchronization bar** - Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with **a thick vertical or horizontal line**.



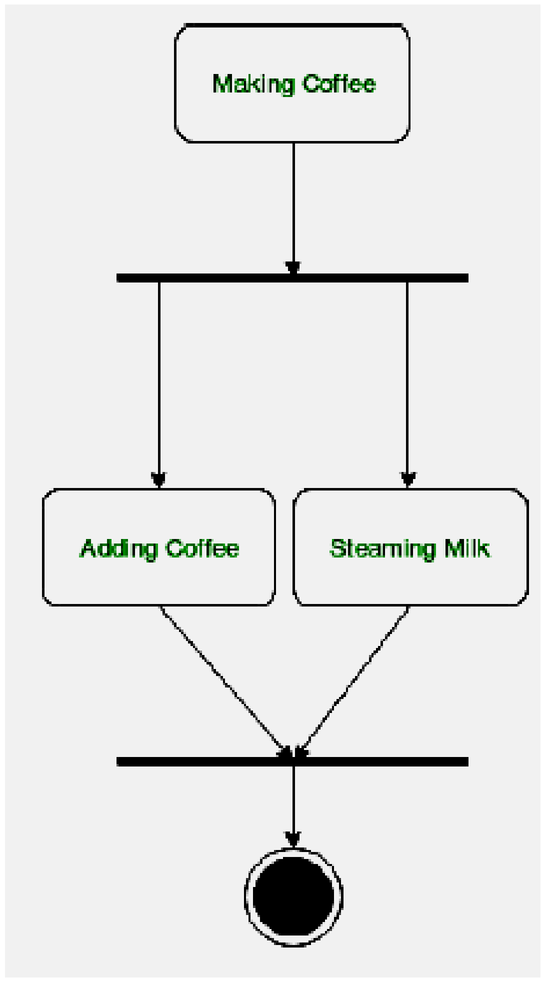
- **Fork symbol** - Splits a single activity flow into two concurrent activities. Symbolized with **multiple arrowed lines from a join**.



- **Loop Node** - A structured activity node that represents a **loop with setup, test, and body sections**

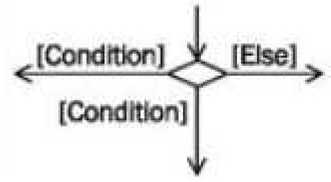


- Fork join example

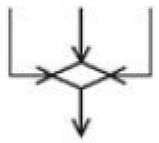


Notation

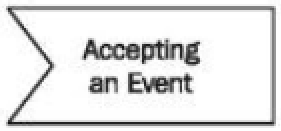
- **Decision Node** - Represents a conditional branch point or **decision** node. A decision node has one input and two or more outputs.
  - Guards are a statement written next to a decision diamond in [....]



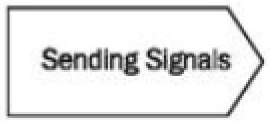
- **Merge Node** - Merges the control flows into a single one. The diamond symbol has several inputs and only one output.



- **Accepting an Event** (Action) - This action waits for an event to occur. After the event is accepted, the flow that comes from this action is executed.



- **Sending Signals** (Action) - Sending a signal means that a signal is being sent to an accepting activity.

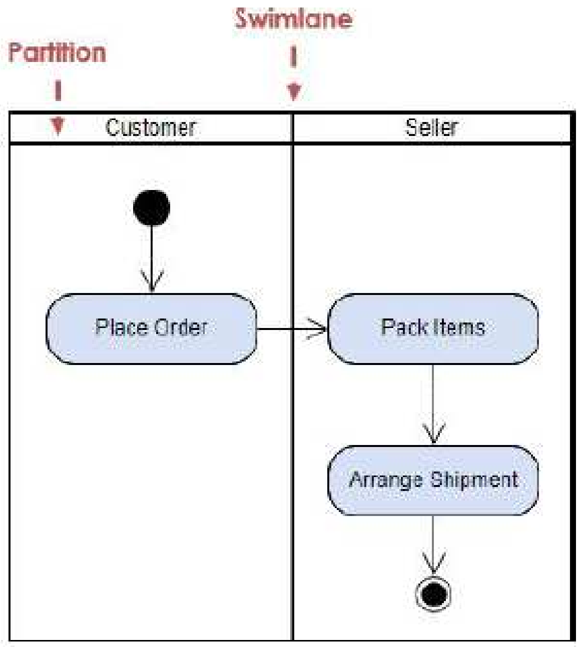


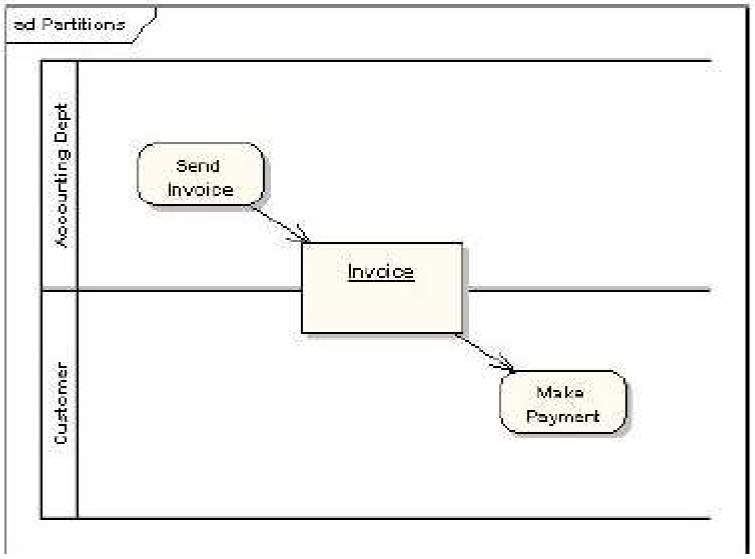
Activity diagram(contd.)

- Activity diagrams support description of parallel activities and synchronization aspects involved in different activities.
- **Parallel activities** are represented on an activity diagram by using swim lanes.
  - Swim lanes enable you to group activities based on who is performing them, e.g., academic department vs. hostel office.
  - Thus swim lanes **subdivide activities** based on the responsibilities of some components.

Notation-Swimlanes

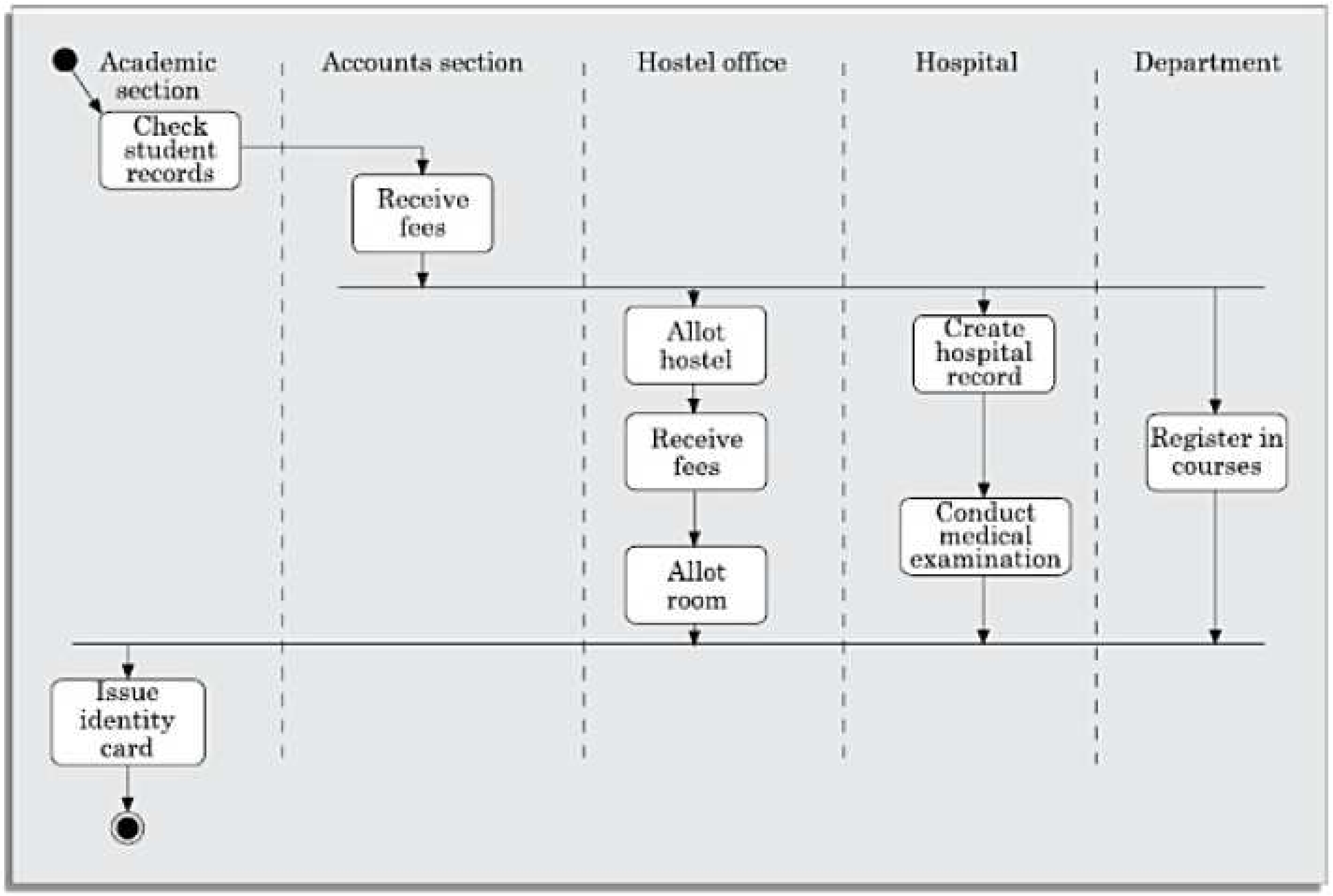
- **Swimlane** and **Partitions-** A way to **group activities(parallel activities) performed by the same actor** on an activity diagram or **to group activities in a single thread**.
  - Swimlanes are used **for grouping related activities in one column/row**
  - Swimlanes can be **vertical** or **horizontal**, it adds modularity to the diagram

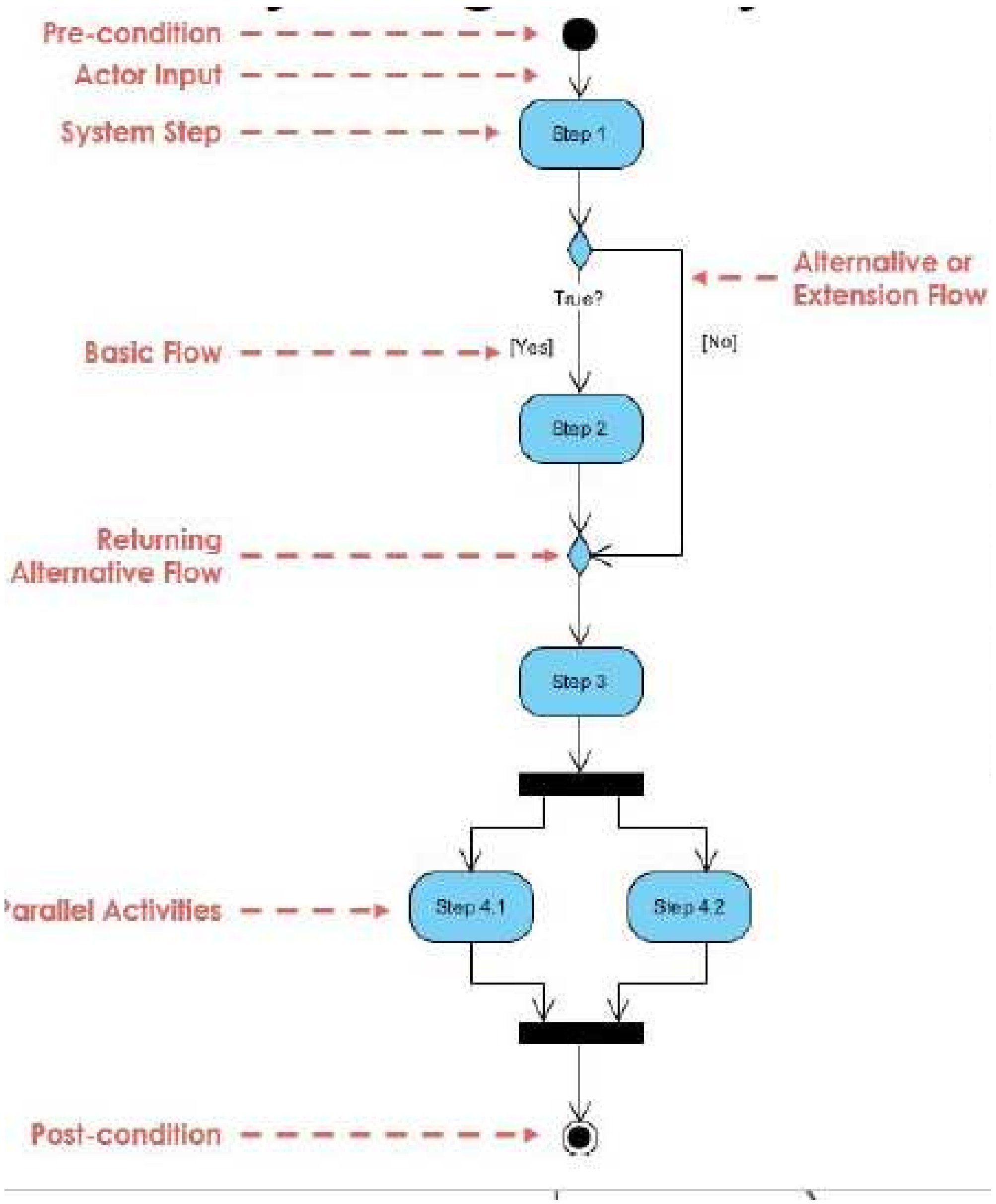


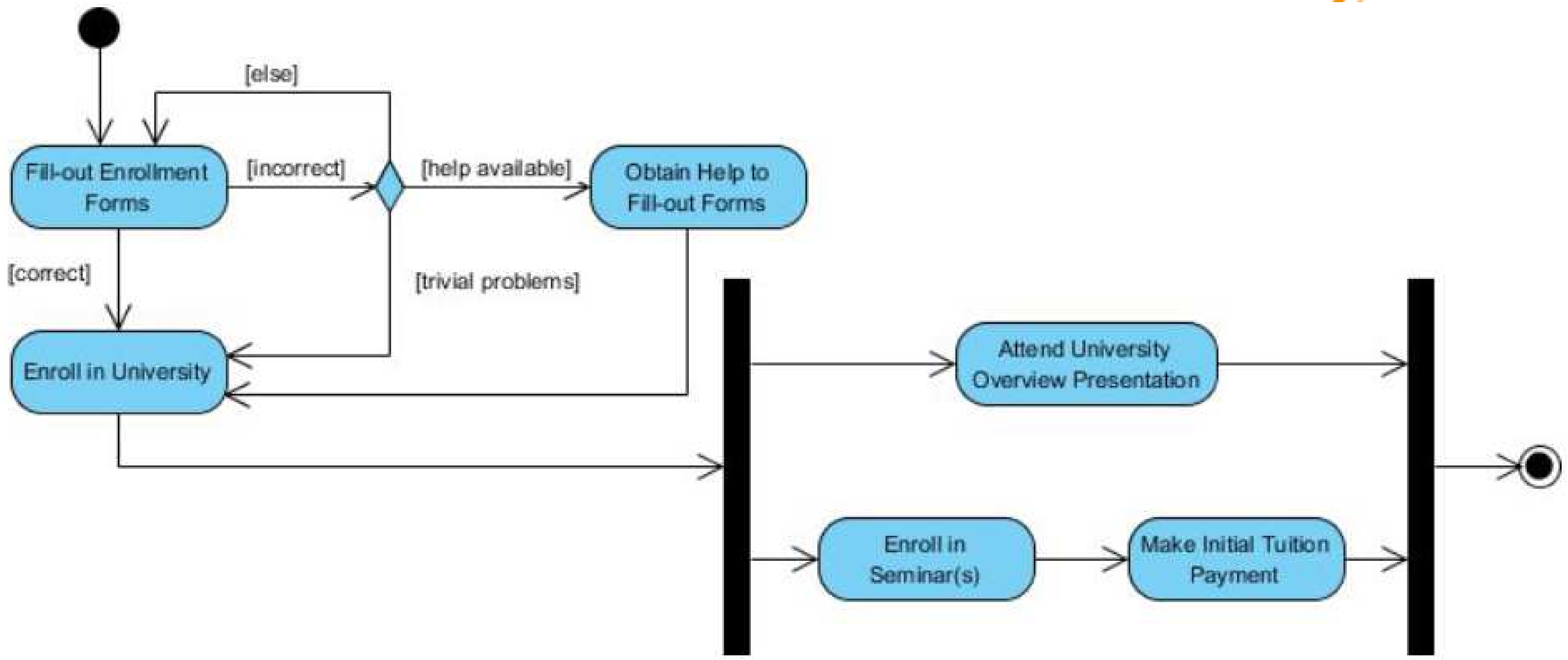


Prepared by Renetha J.B.

Example –student admission

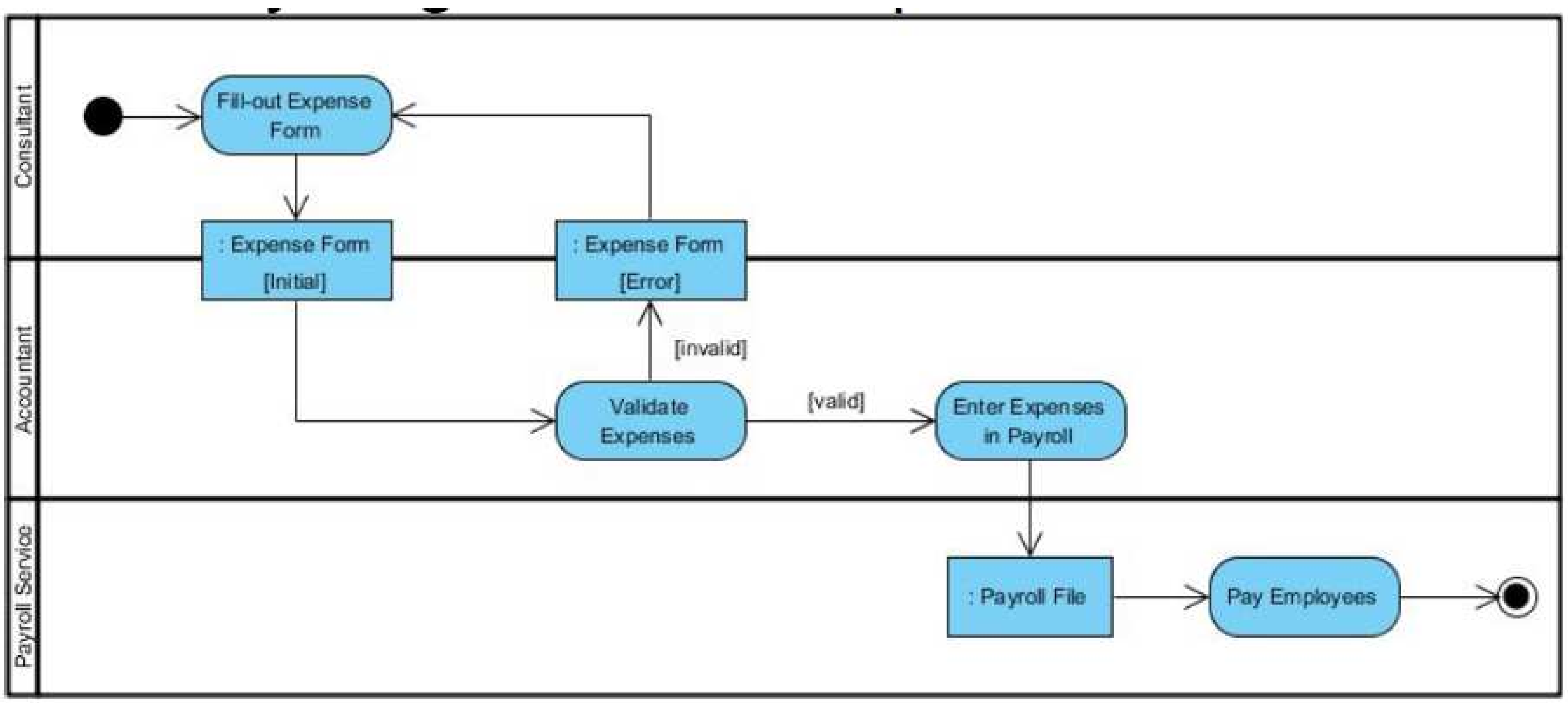



Pre-condition
Actor Input
System Step
Step 1
True?
Alternative or Extension Flow
[No]
Basic Flow
[Yes]
Step 2
Returning Alternative Flow
Step 3
'arallel Activities
Step 4.1
Step 4.2
Post-condition


[else]
Fill-out Enrollment Forms
[incorrect]
[help available]
Obtain Help to Fill-out Forms
[correct]
[trivial problems]
Enroll in University
Attend University Overview Presentation
Enroll in Seminar(s)
Make Initial Tuition Payment

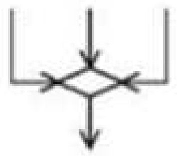
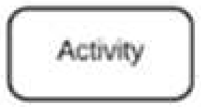
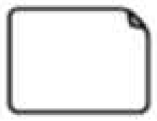
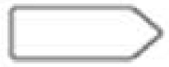
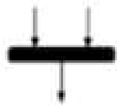
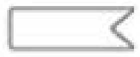
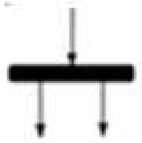
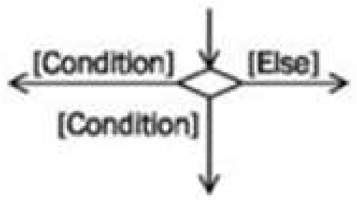
- The applicant hands a filled out copy of Enrollment Form.
- The registrar inspects the forms.
- The registrar determines that the forms have been filled out properly.
- The registrar informs student to attend in university overview presentation.
- The registrar helps the student to enroll in seminars
- The registrar asks the student to pay for the initial tuition.

Activity diagram - E.g.



- Staff expense subimssion.

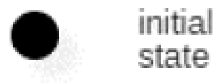
Activity diagram symbols

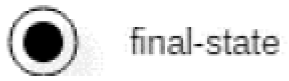
●	Start symbol		Merge Node symbol
	Activity symbol		Note symbol
	Connector symbol		send signal symbol
	Joint symbol/ Synchronization bar		Receive signal symbol
	Fork symbol	⊗	Flow final symbol
[condition]	Condition text (guard)	◉	End symbol
	Decision Node		

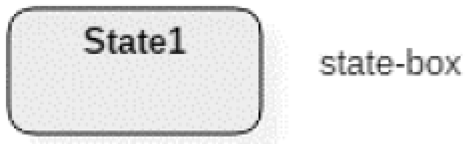
Statechart diagram

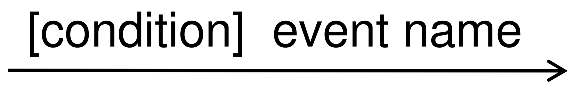
- A state chart diagram is normally used to model **how the state of an object changes in its life time.**
- If we are interested in modelling some behaviour that involves several objects collaborating with each other, state chart diagram is **not** appropriate. Use sequence or collaboration diagrams.
- State chart diagrams are based on the finite state machine (FSM) formalism.
  - In FSM, the number of states becomes too many and the model too complex when used to model practical systems.
  - This problem is overcome in UML by using state charts.

Basic elements of a state chart

- The basic elements of the state chart diagram are as follows:
  - **Initial state:** This represented as a **filled circle**. 

  - **Final state:** This is represented by a **filled circle inside a larger circle.** 

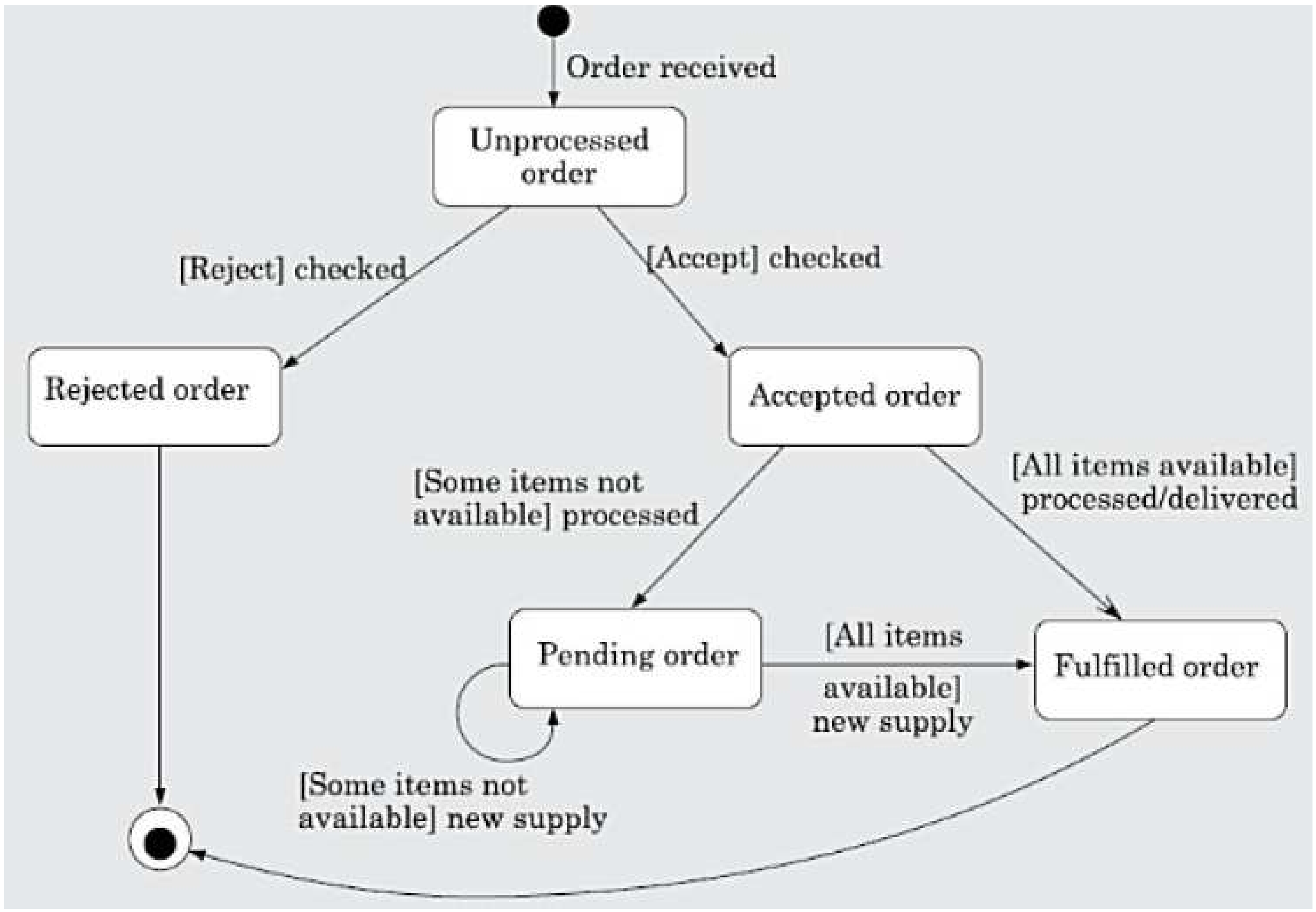
  - **State:** These are represented **by rectangles with rounded corners.** 

- **Transition:** A transition is shown as an **arrow between two states.**
  - Normally, the <u>name of the event</u> which causes the transition is places <u>along side the arrow</u>. We can also assign a guard(condition) to the transition 

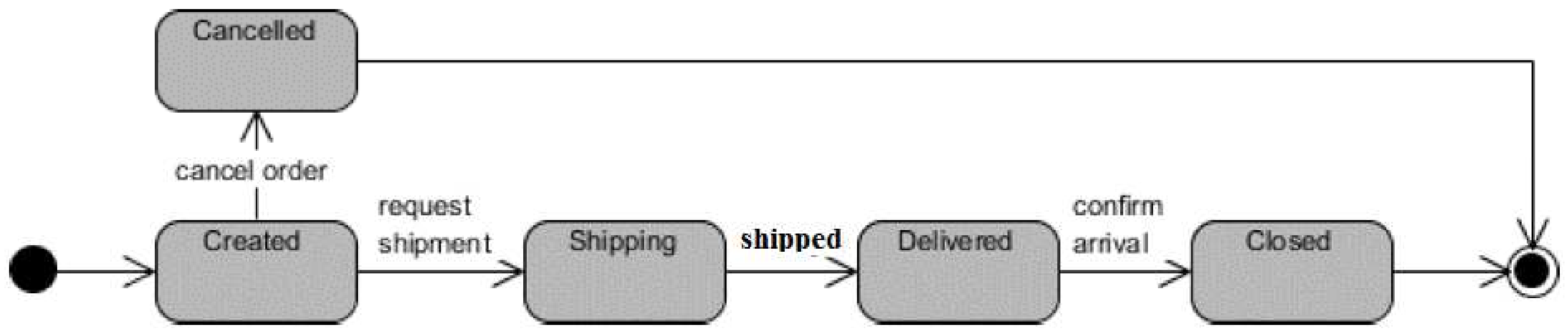
State chart diagram(contd.)

- A state chart is a hierarchical model of a system and introduces the concept of a **composite state** (also called **nested state**).
- Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer.
- An activity can be interrupted by an event.
- **Activity diagram** is flow of ***activities without trigger*** (event) mechanism, **state chart diagram** consist of **triggered states(driven by event)**

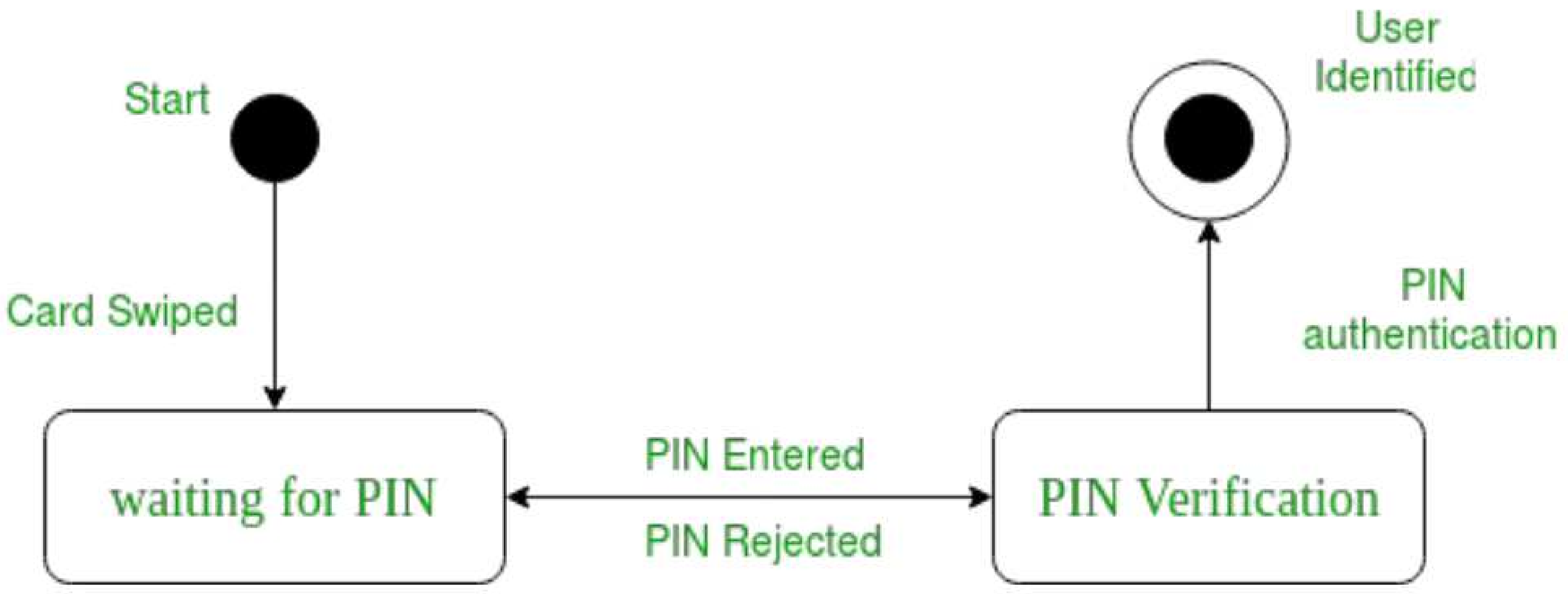
State chart-(Example)-Order object


Order received
Unprocessed order
[Reject] checked
[Accept] checked
Rejected order
Accepted order
[Some items not available] processed
[All items available] processed/delivered
Pending order
[All items available] new supply
Fulfilled order
[Some items not available] new supply

State chart-(Example)-Order object

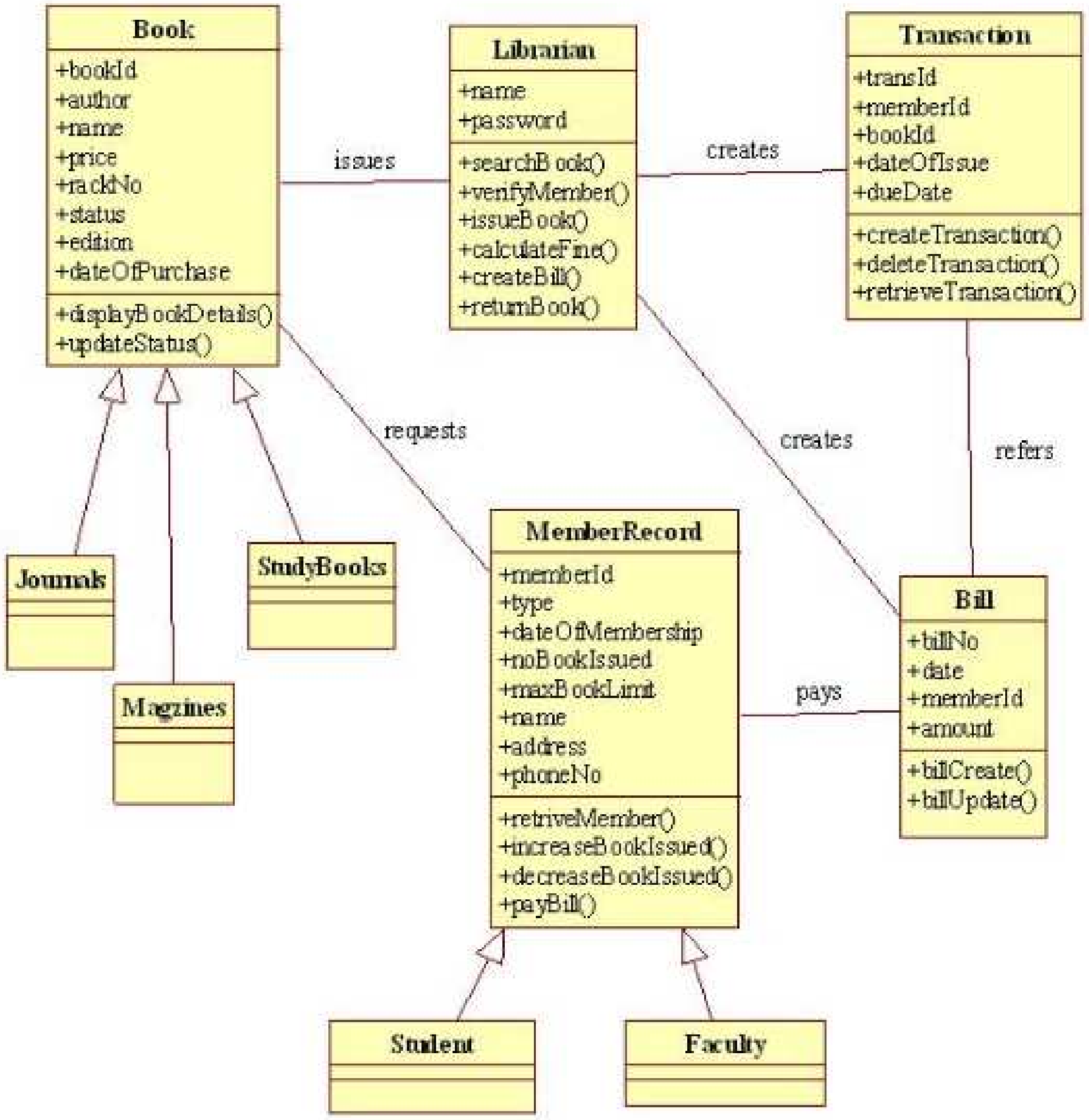


State chart diagram-ATM



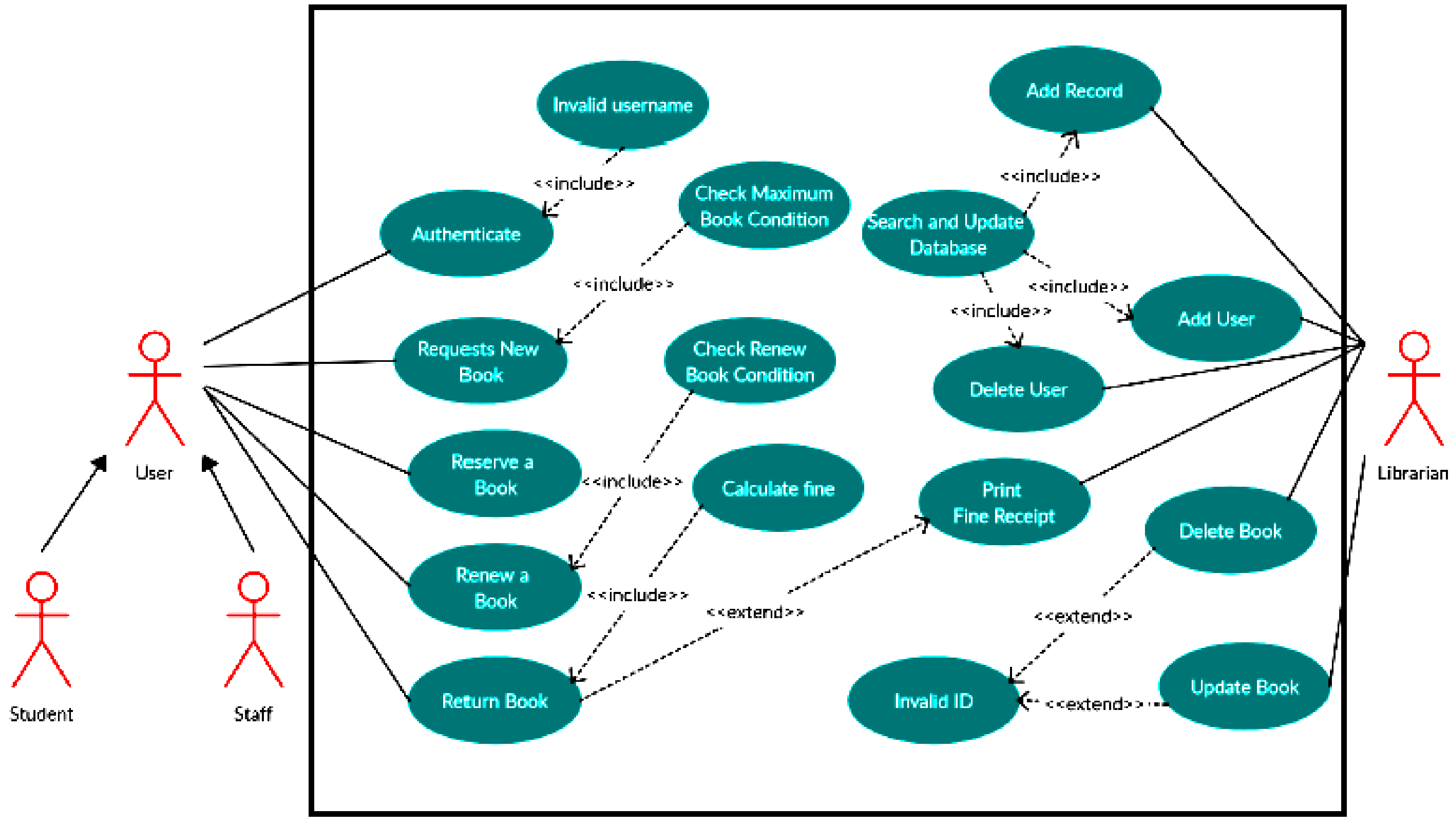
EXAMPLES

CLASSdiagram- Library

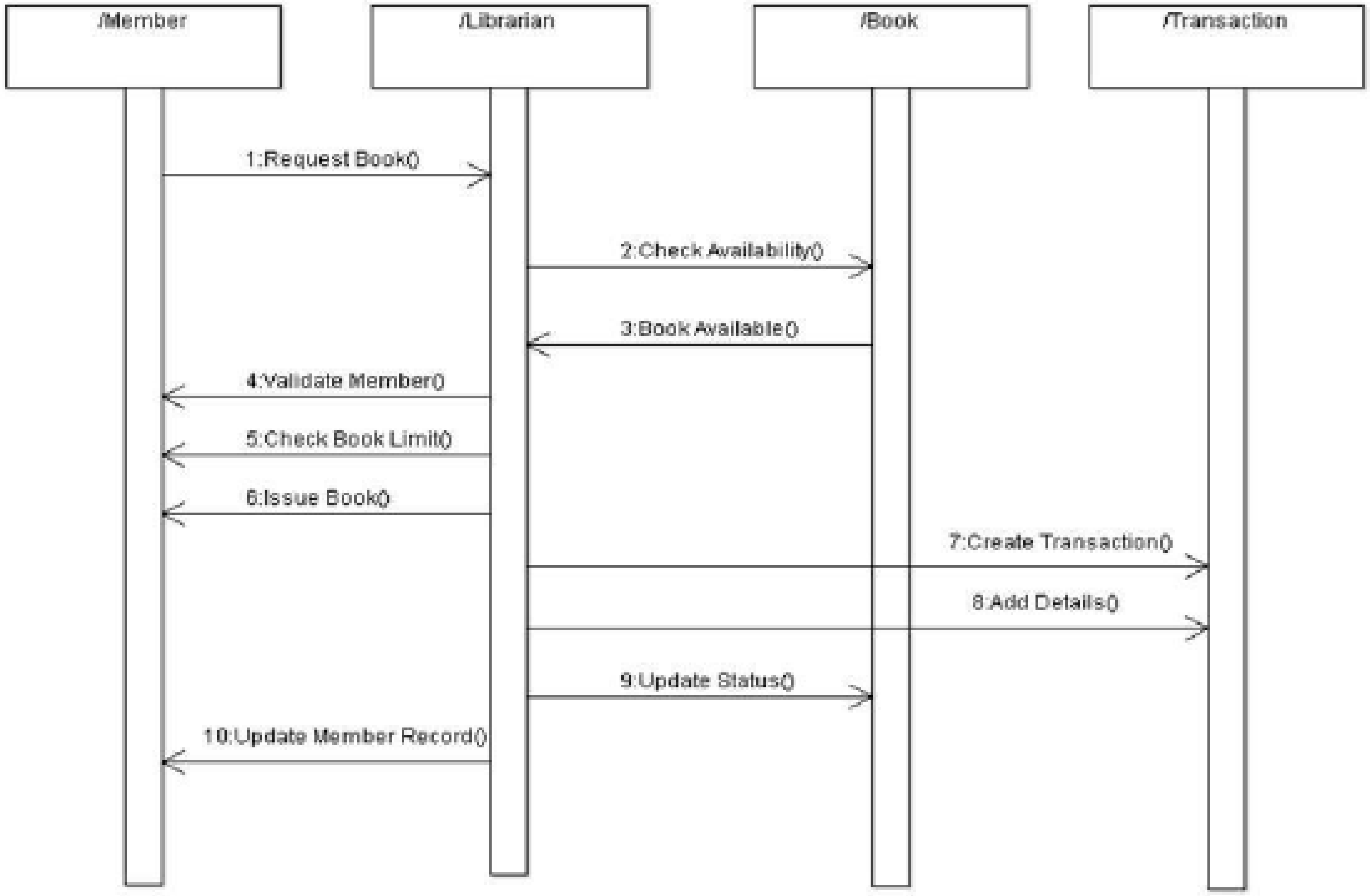


Prepared by Renetha J.B.

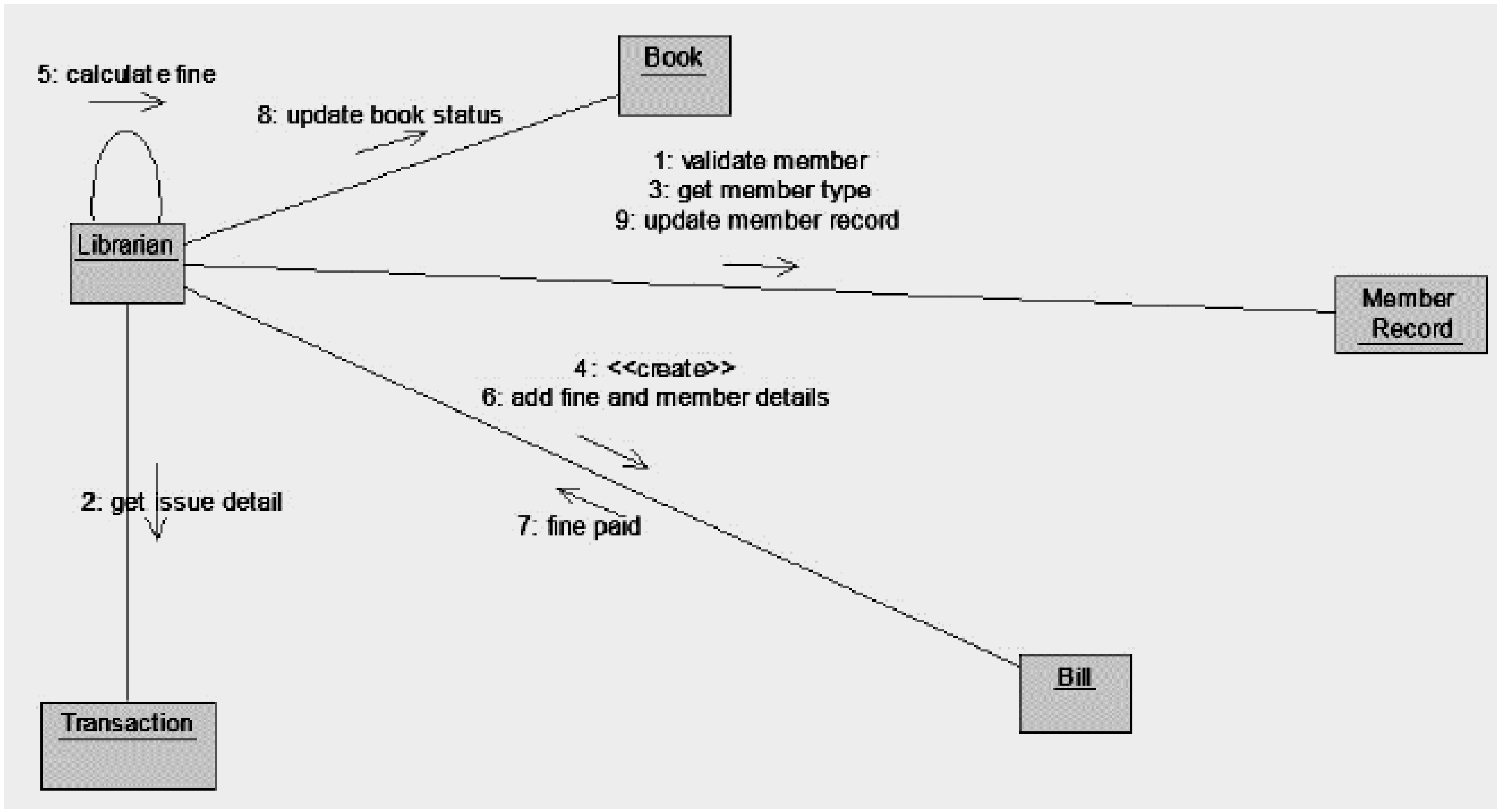
Library--Use case diagram



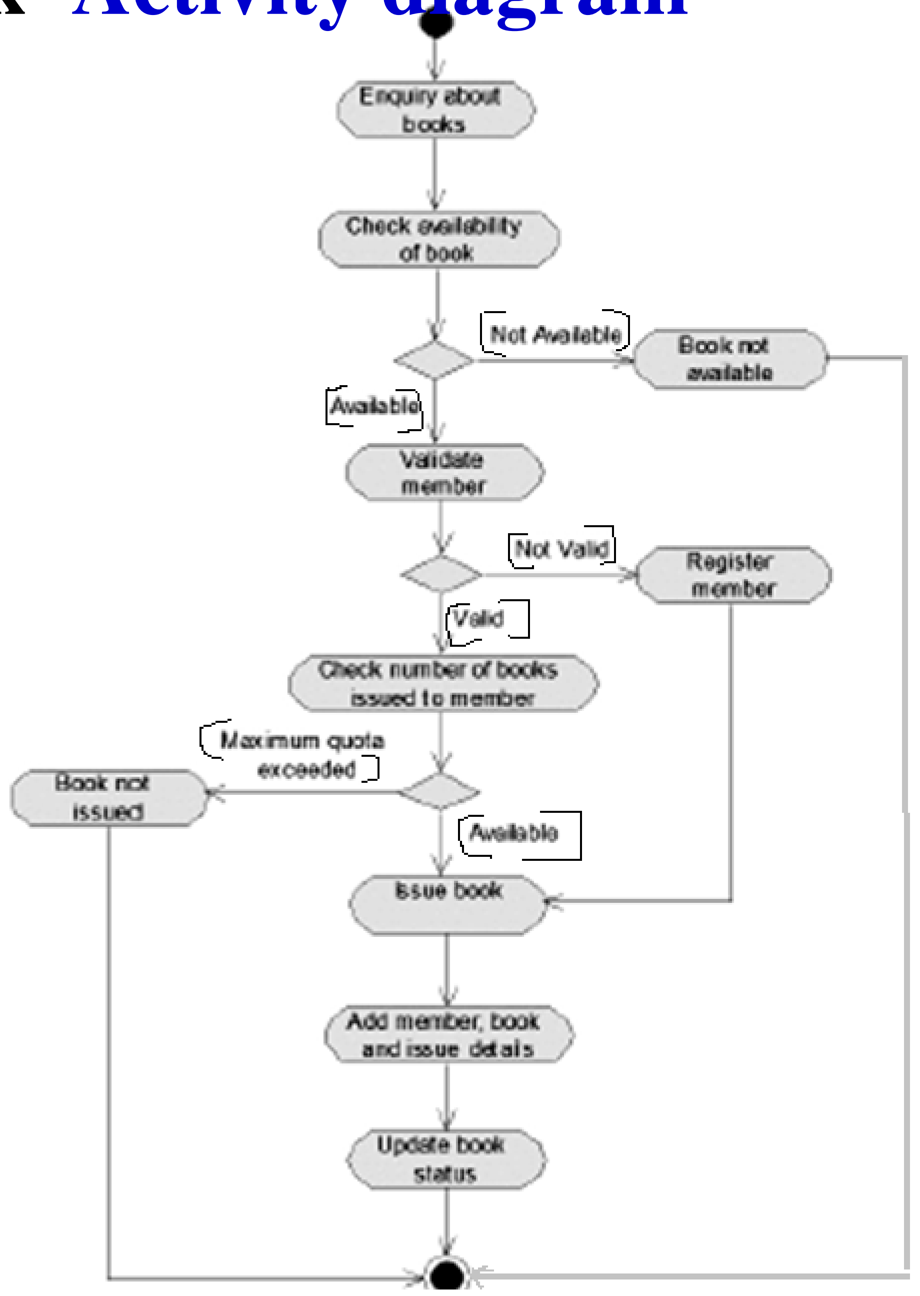
Book issue- Library—Sequence diagram



Library-Collaboration diagram

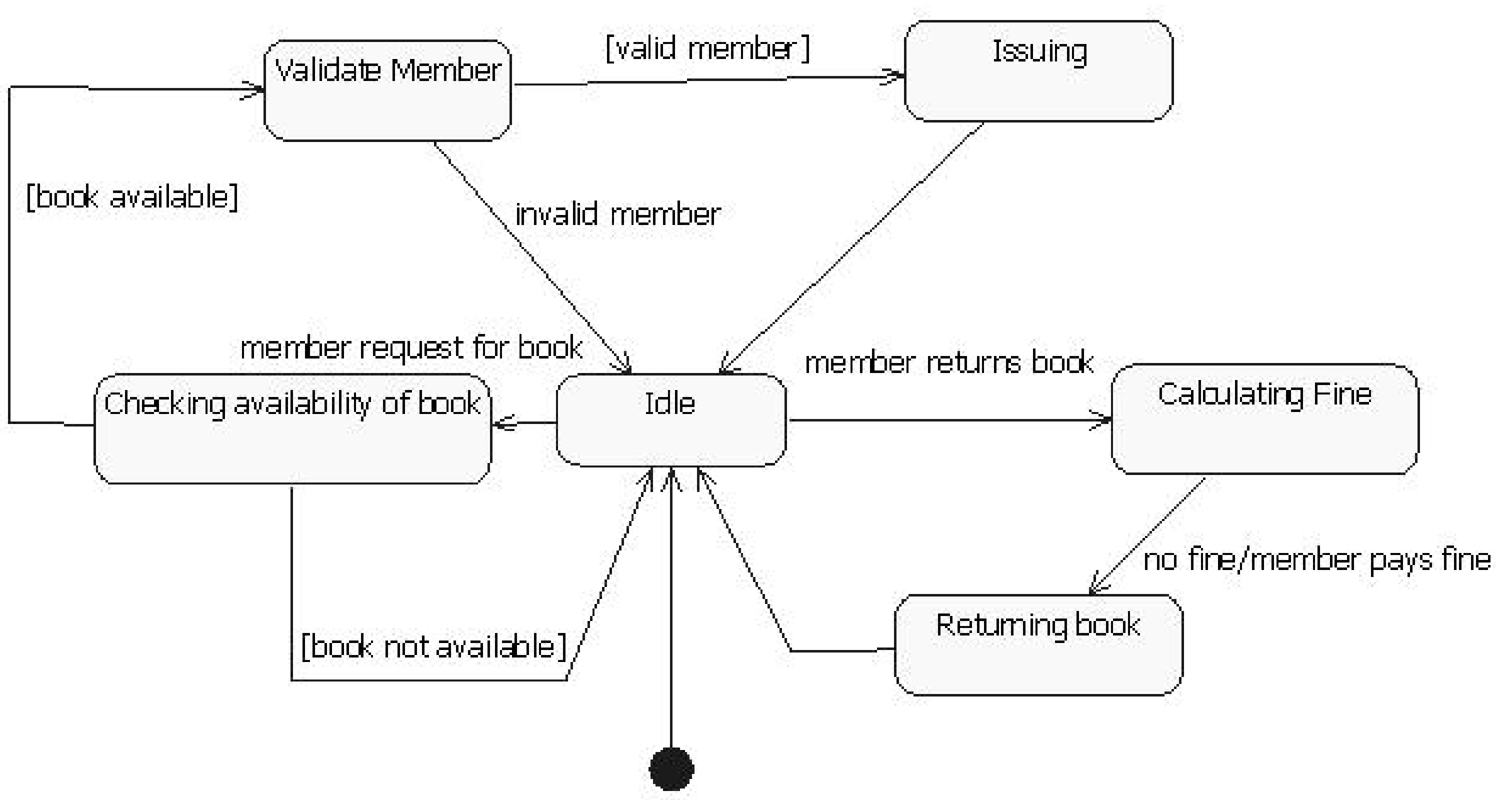


Library Issue book- Activity diagram



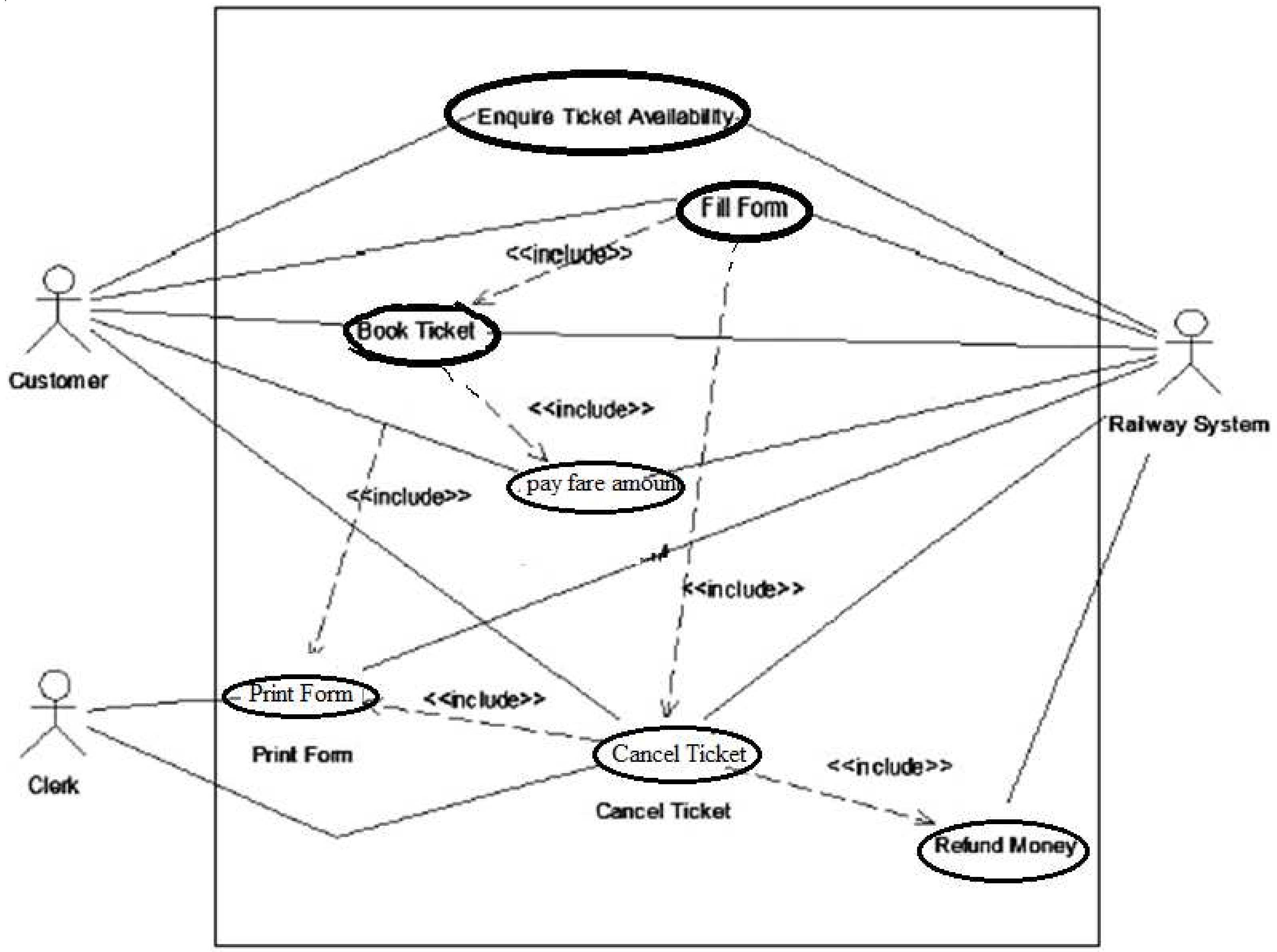
Prepared by Renetha J.B.

Library-State chart diagram



Prepared by Renetha J.B.

Railway System-Use case diagram



Reference

- Rajib Mall, **Fundamentals of Software Engineering, 4th edition, PHI, 2014**